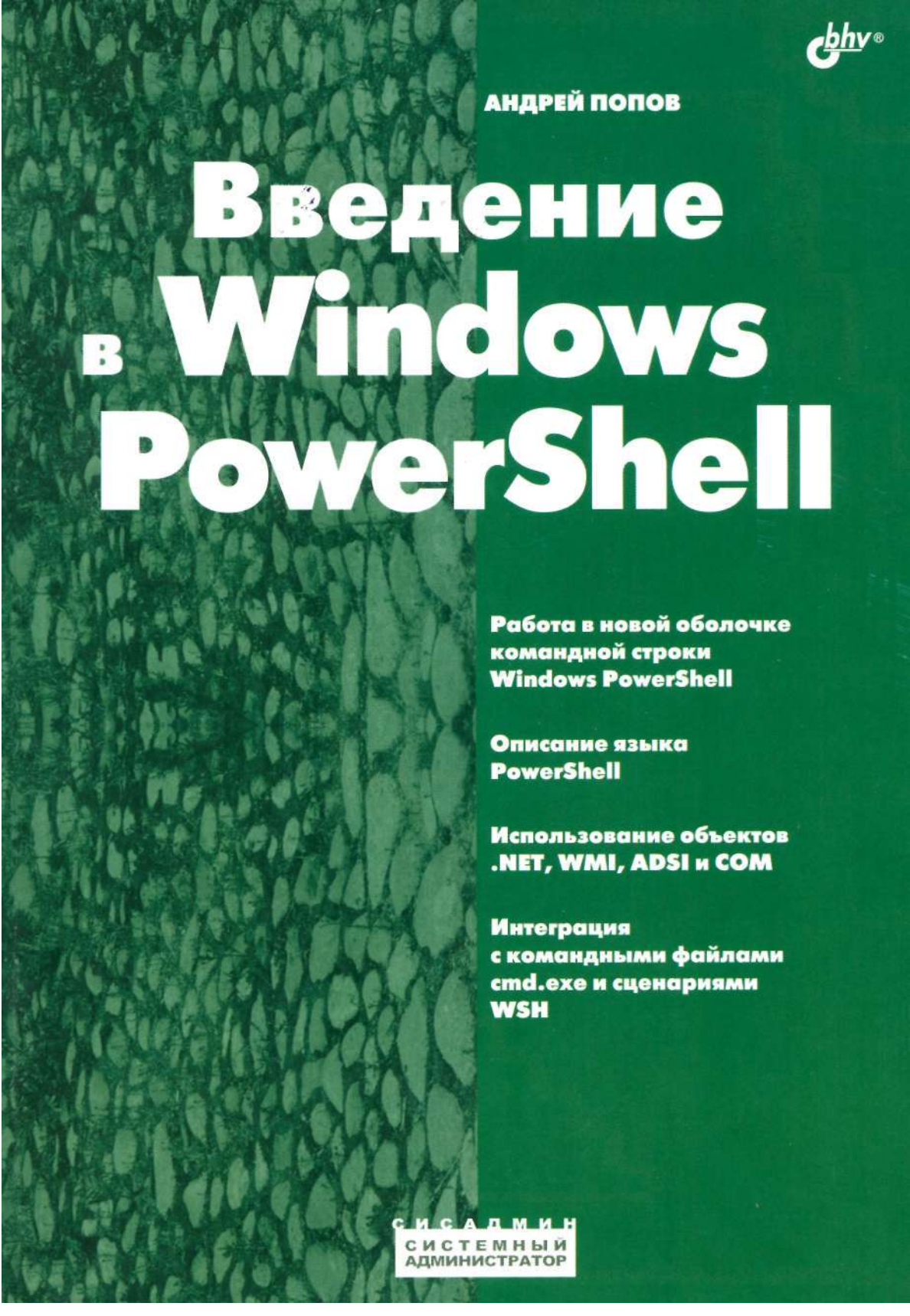
VISIT…


bhv®
АНДРЕЙ ПОПОВ
Введение в Windows PowerShell
Работа в новой оболочке командной строки Windows PowerShell
Описание языка PowerShell
Использование объектов .NET, WMI, ADSI и COM
Интеграция с командными файлами cmd.exe и сценариями WSH
СИСАДМИН
СИСТЕМНЫЙ АДМИНИСТРАТОР

УДК 681.3.06
ББК 32.973.26-018.2
П58

Попов А. В.

П58 Введение в Windows PowerShell. — СПб.: БХВ-Петербург, 2009. — 464 с.: ил. — (Системный администратор)

ISBN 978-5-9775-0283-2

Рассматривается новая объектно-ориентированная оболочка командной строки Microsoft Windows PowerShell и ее возможности для автоматизации повседневных задач администрирования. Описываются основные элементы и конструкции языка PowerShell. Приводятся примеры использования объектов .NET, WMI, ADSI и COM. Обсуждаются вопросы совместного использования PowerShell, командных файлов интерпретатора cmd.exe и сценариев Windows Script Host. Даются примеры решения с помощью PowerShell задач администратора Windows.

Для администраторов, программистов и опытных пользователей Windows

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Якубович*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.08.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 37,41.
Тираж 2000 экз. Заказ № 541
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0283-2

© Попов А. В., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Оглавление

Введение **1**

Что это за книга и для кого она предназначена 2
Структура книги 3
Принятые в книге соглашения 5

Часть I. Изучаем PowerShell **7**

Глава 1. Windows PowerShell — результат развития технологий автоматизации **9**

Предшественники PowerShell в мире Windows 9
Оболочка командной строки command.com/cmd.exe 12
Сервер сценариев Windows Script Host (WSH) 14
Оболочка WMI Command-line (WMIC) 17
Причины и цели создания оболочки PowerShell 18
Отличие PowerShell от других оболочек — ориентация на объекты 22

Глава 2. Первые шаги в PowerShell. Основные понятия **25**

Загрузка и установка PowerShell 25
Запуск оболочки 26
Работают ли знакомые команды? 26
Вычисление выражений 28
Типы команд PowerShell 29
Командлеты 30
Функции 35
Сценарии 36
Внешние исполняемые файлы 36
Псевдонимы команд 36
Диски PowerShell 40
Провайдеры PowerShell 41

Навигация по дискам PowerShell ... 43
Просмотр содержимого дисков и каталогов ... 44
Создание дисков ... 46

Глава 3. Приемы работы в оболочке ... 48

Редактирование в командном окне PowerShell ... 48
Автоматическое завершение команд ... 50
Справочная система PowerShell ... 52
Получение справки о командлетах ... 53
Справочная информация, не связанная с командлетами ... 55
История команд в сеансе работы ... 58
Протоколирование действий в сеансе работы ... 62

Глава 4. Настройка оболочки ... 65

Настройка ярлыка PowerShell ... 65
Программное изменение свойств консоли PowerShell ... 67
Цвета текста и фона ... 68
Заголовок командного окна ... 68
Размеры командного окна ... 69
Приглашение командной строки ... 70
Настройка пользовательских профилей ... 72
Политики выполнения сценариев ... 75

Глава 5. Работа с объектами ... 78

Конвейеризация объектов в PowerShell ... 78
Просмотр структуры объектов (командлет *Get-Member*) ... 80
Фильтрация объектов (командлет *Where-Object*) ... 82
Сортировка объектов (командлет *Sort-Object*) ... 85
Выделение объектов и свойств (командлет *Select-Object*) ... 87
Выполнение произвольных действий над объектами в конвейере (командлет *ForEach-Object*) ... 90
Группировка объектов (командлет *Group-Object*) ... 91
Измерение характеристик объектов (командлет *Measure-Object*) ... 92
Вызов статических методов ... 93
Управление выводом команд в PowerShell ... 95
Форматирование выводимой информации ... 96
Перенаправление выводимой информации ... 99

Глава 6. Переменные, массивы и хэш-таблицы ... 102

Числовые и символьные литералы ... 102
Числовые литералы ... 102
Символьные строки ... 103

Переменные PowerShell ... 107
Переменные оболочки PowerShell ... 108
Пользовательские переменные. Типы переменных ... 111
Переменные среды Windows ... 115
Массивы в PowerShell ... 117
Обращение к элементам массива ... 117
Операции с массивом ... 118
Хэш-таблицы (ассоциативные массивы) ... 121
Операции с хэш-таблицей ... 124

Глава 7. Операторы и управляющие инструкции ... 126

Арифметические операторы ... 126
Оператор сложения ... 127
Оператор умножения ... 129
Операторы вычитания, деления и остатка от деления ... 130
Операторы присваивания ... 131
Операторы сравнения ... 133
Операторы проверки на соответствие шаблону ... 135
Логические операторы ... 138
Управляющие инструкции языка PowerShell ... 139
Инструкция *If ... ElseIf ... Else* ... 139
Цикл *While* ... 140
Цикл *Do ... While* ... 141
Цикл *For* ... 141
Цикл *ForEach* ... 142
Метки циклов, инструкции *Break* и *Continue* ... 145
Инструкция *Switch* ... 146

Глава 8. Функции, фильтры и сценарии ... 152

Функции в PowerShell ... 152
Обработка аргументов функций с помощью переменной *$Args* ... 153
Формальные параметры функций ... 155
Возвращаемые значения ... 160
Функции внутри конвейера команд ... 162
Фильтры в PowerShell ... 163
Функции в качестве командлетов ... 164
Сценарии PowerShell ... 166
Создание и запуск сценариев PowerShell ... 167
Передача аргументов в сценарии ... 169
Выход из сценариев ... 170
Оформление сценариев. Комментарии ... 171

Глава 9. Обработка ошибок и отладка ... 175

Обработка ошибок ... 175
Объект ErrorRecord и поток ошибок ... 176
Сохранение объектов, соответствующих ошибкам ... 179
Мониторинг возникновения ошибок ... 182
Режимы обработок ошибок ... 183
Обработка "критических" ошибок (исключений) ... 185
Отладка сценариев ... 187
Вывод диагностических сообщений ... 187
Командлет Set-PSDebug ... 188
Трассировка выполнения команд ... 189
Пошаговое выполнение команд ... 191
Вложенная командная строка и точки прерывания ... 191

Часть II. Используем PowerShell ... 195

Глава 10. Доступ из PowerShell к внешним объектам (COM, WMI, .NET и ADSI) ... 197

Работа с COM-объектами ... 197
Внешние серверы автоматизации на примере Microsoft Office ... 202
Доступ к объектам WMI ... 208
Подключение к подсистеме WMI. Получение списка классов ... 209
Получение объектов WMI ... 211
Выполнение WQL-запросов ... 214
Использование объектов .NET ... 215
Доступ к службе каталогов ADSI ... 218

Глава 11. Работа с файловой системой ... 222

Навигация в файловой системе ... 222
Получение списка файлов и каталогов ... 223
Определение размера каталогов ... 227
Создание файлов и каталогов ... 228
Чтение и просмотр содержимого файлов ... 229
Запись файлов ... 230
Копирование файлов и каталогов ... 232
Переименование и перемещение файлов и каталогов ... 235
Удаление файлов и каталогов ... 236
Поиск текста в файлах ... 237
Замена текста в файлах ... 241

Глава 12. Управление процессами и службами ... 243

Управление процессами ... 243
Просмотр списка процессов ... 245
Определение библиотек, используемых процессом ... 250
Остановка процессов ... 252
Запуск процессов, изменение приоритетов выполнения ... 253
Завершение неотвечающих процессов ... 255
Управление службами ... 255
Просмотр списка служб ... 256
Остановка и приостановка служб ... 258
Запуск и перезапуск служб ... 259
Изменение параметров службы ... 259

Глава 13. Работа с системным реестром ... 261

Структура реестра ... 261
Просмотр локального реестра ... 263
Просмотр удаленного реестра ... 265
Модификация реестра ... 268
Создание нового раздела ... 269
Копирование разделов ... 269
Переименование раздела ... 270
Удаление раздела ... 270
Создание параметра ... 270
Изменение значения параметра ... 271
Переименование параметра ... 272
Копирование параметров ... 272
Очистка значения параметра ... 273
Удаление параметра ... 273

Глава 14. Работа с журналами событий ... 275

Инструменты для обработки журналов событий ... 277
Список журналов событий на локальном компьютере ... 279
Список журналов событий на удаленном компьютере ... 281
Просмотр событий из локального журнала ... 282
Вывод событий определенного типа ... 285
Отбор событий по идентификатору ... 286
Отбор событий по датам ... 287
Группировка событий по источнику возникновения ... 288
Просмотр событий из удаленного журнала ... 289

Настройка журналов событий ... 292
Установка максимального размера журналов ... 292
Установка режима хранения журналов ... 292
Очистка журнала ... 293

Глава 15. Управление рабочими станциями. Получение и анализ системной информации ... 295

Завершение сеанса пользователя ... 295
Перезагрузка и выключение компьютера ... 297
Получение информации о BIOS ... 298
Вывод списка команд, выполняемых при загрузке системы ... 299
Вывод свойств операционной системы ... 301
Вывод списка установленных программных продуктов ... 303
Вывод списка установленных обновлений операционной системы ... 306

Глава 16. Инвентаризация оборудования ... 309

Получение информации о физической памяти ... 309
Преобразование отчета в формат HTML ... 311
Получение информации о процессорах ... 313
Получение списка устройств Plug-and-Play ... 316
Получение информации о звуковой карте ... 319
Получение информации о видеокарте ... 320
Получение информации о сетевых адаптерах ... 323

Глава 17. Настройка сетевых параметров. Работа с электронной почтой ... 325

Получение и настройка сетевых параметров ... 325
Получение списка IP-адресов компьютера ... 326
Вывод параметров протокола TCP/IP ... 327
Настройка DHCP ... 332
Отправка сообщений по электронной почте ... 338

Глава 18. PowerShell, cmd.exe и VBScript: совместное использование ... 341

Сравнение языков PowerShell и cmd.exe ... 342
Различия в синтаксисе команд ... 342
Работа с переменными ... 344
Использование циклов ... 346
Вывод текста и запуск программ ... 347
Запуск команд cmd.exe из PowerShell ... 347

Сравнение языков PowerShell и VBScript ... 349
Обращение к функциям, командам и методам ... 349
Работа с переменными, массивами и объектами ... 351
Использование символьных строк ... 351
Прочие замечания по синтаксису ... 352
Аналоги PowerShell для функций VBScript ... 353
Математические функции ... 353
Символьные функции ... 355
Функции для работы с датами и временем ... 361
Использование из PowerShell кода VBScript ... 368
Использование из PowerShell кода JScript ... 370

Заключение ... 371

ПРИЛОЖЕНИЯ ... 373

Приложение 1. Объектная модель WMI ... 375

Общая структура WMI ... 376
Ядро WMI ... 377
Провайдеры WMI ... 378
Менеджер объектов CIM ... 379
Репозиторий CIM. Пространства имен ... 381
Путь к классам и объектам CIM ... 384
Безопасность при работе с WMI ... 385
Структура классов WMI ... 389
Основные типы классов CIM ... 389
Свойства классов WMI ... 391
Методы классов WMI ... 397
Квалификаторы классов, свойств и методов ... 399
Интерактивная работа с объектами WMI ... 403
Тестер WMI (WBEMTest) ... 403
Административные утилиты WMI (WMI Tools) ... 404

Приложение 2. Полезные COM-объекты и примеры их использования ... 411

Управление проводником Windows с помощью объекта *Shell.Application* ... 411
Отображение специальных окон Проводника ... 413
Вызов элементов панели управления ... 418
Управление открытыми окнами ... 420

Использование объектов Windows Script Host 423
Работа с ресурсами локальной сети (объект *WScript.Network*) 423
Вывод информационного окна (объект *WScript.Shell*) 428
Переключение между приложениями, имитация нажатий клавиш (объект *WScript.Shell*) 430
Доступ к специальным папкам Windows (объект *WScript.Shell*) 436
Удаление некорректных ярлыков (объект *WScript.Shell*) 438

Ссылки на ресурсы Интернета 441

Сайты компании Microsoft 441
Другие сайты 441
Группы новостей 442
Блоги 442

Список литературы 443

Предметный указатель 445

*

Введение

Windows PowerShell — это новая оболочка командной строки и среда выполнения сценариев для операционной системы Windows, разработанная компанией Microsoft относительно недавно (финальный релиз версии 1.0 этой оболочки был выпущен в ноябре 2006 года). Главной задачей, которую ставили перед собой разработчики, было создание среды составления сценариев, которая наилучшим образом подходила бы для современных версий операционной системы Windows и была бы более функциональной, расширяемой и простой в использовании, чем какой-либо аналогичный продукт для любой другой операционной системы. В первую очередь эта среда должна была подходить для решения задач, стоящих перед системными администраторами, а также удовлетворять требованиям разработчиков программного обеспечения, предоставляя им средства для быстрой реализации интерфейсов управления создаваемыми приложениями. Сейчас уже можно сказать, что продукт оказался очень продуманным и удачным, а его мощные возможности сочетаются с простотой использования. Актуальность изучения и использования PowerShell вызвана тем, что компания Microsoft в настоящее время позиционирует эту оболочку как основной инструмент управления операционной системой и рядом разработанных ею приложений (PowerShell официально включен в качестве стандартного компонента в операционную систему Windows Server 2008, а также используется в таких продуктах Microsoft, как Exchange Server 2007, System Center Operations Manager 2007, System Center Data Protection Manager V2 и System Center Virtual Machine Manager).

За рубежом выпущено уже несколько книг, посвященных различным аспектам PowerShell (отдельно выделим здесь электронные книги [8] и [9], которые можно свободно загрузить из Интернета). Также много ресурсов имеется в Интернете, отметим некоторые из них (адреса этих ресурсов приведены в конце книги в *разд. "Ссылки на ресурсы Интернета"*):

- на сайте Microsoft поддерживается активно обновляемый раздел TechNet Script Center, содержащий примеры сценариев PowerShell, ссылки на статьи и вебкасты, посвященные этому продукту;

- разработчики PowerShell ведут в Интернете блоги, в которых они делятся информацией о возможностях PowerShell;
- действует группа новостей, посвященная PowerShell.

К сожалению, с систематизированной информацией о PowerShell на русском языке дело обстоит несколько хуже, особенно не хватает подробного описания нового языка программирования PowerShell. Что можно сейчас найти в Интернете по этой теме? На сайте компании Microsoft доступны локализованная версия PowerShell (встроенная справка переведена на русский язык) и пакет документации на русском языке (введение в новую оболочку командной строки и краткий обзор языка PowerShell). Периодически переводы статей по PowerShell появляются в журналах "Windows & .Net Magazine/RE" (**http://www.osp.ru**) и "TechNet Magazine" (**http://www.microsoft.com/technet/technetmag**). Отдельно хочется сказать спасибо Д. Сотникову, А. Бешкову и В. Гусеву, ведущим блоги и проводящим вебкасты по тематике PowerShell.

Что это за книга и для кого она предназначена

При написании этой книги была поставлена цель восполнить недостаток информации о PowerShell на русском языке. В частности, хотелось решить следующие задачи:

- пояснить лежащие в основе PowerShell базовые механизмы работы с объектами и описать основные конструкции и элементы языка PowerShell (за основу взята великолепно написанная монография [1], которую можно порекомендовать всем, кто захочет более глубоко изучить PowerShell);
- помочь пользователям, знакомым с командным интерпретатором cmd.exe и сервером сценариев WSH, начать работу с новым средством автоматизации Windows;
- привести практические примеры использования PowerShell для решения типичных задач администратора операционной системы Windows.

Сама постановка этих задач подразумевает некоторое знакомство читателя с компьютером. Поэтому книга ориентирована на администраторов информационных систем на базе Windows и обычных опытных пользователей, желающих изучить новую мощную оболочку командной строки от Microsoft и автоматизировать с помощью нее свою повседневную работу на компьютере.

Структура книги

Книга состоит из двух частей. В первой части, озаглавленной *"Изучаем PowerShell"*, последовательно рассматриваются основные концепции новой оболочки командной строки, элементы и конструкции языка PowerShell.

В *первой главе* обсуждаются возможности и недостатки средств автоматизации работы в операционной системе Windows, существовавших до появления PowerShell (оболочки cmd.exe и WMIC, сервера сценариев Windows Script Host), приводятся причины и цели разработки новой оболочки командной строки. Особо выделяется основное отличие PowerShell от других распространенных оболочек командной строки — ориентация на работу с объектами, а не с потоком текста.

Во *второй главе* приводятся инструкции по установке и запуску PowerShell, описываются типы команд, используемые в данной оболочке. Обсуждаются понятия псевдонимов команд и дисков PowerShell.

В *третьей главе* изучаются приемы интерактивной работы в оболочке PowerShell и способы обращения к встроенной справочной системе.

В *четвертой главе* рассматриваются вопросы настройки интерфейса оболочки, пользовательских профилей и политик выполнения сценариев PowerShell.

В *пятой главе* речь идет об основном механизме PowerShell — конвейеризации объектов, и в качестве примеров описываются базовые манипуляции с объектами, которые можно выполнять в этой оболочке (фильтрация, сортировка, группировка и т. д.). Разбираются механизмы управления выводом команд PowerShell (форматирование и перенаправление результирующей информации).

Шестая глава посвящена изучению основных структур данных, использующихся в PowerShell (константы, переменные, массивы и хэш-таблицы).

В *седьмой главе* рассматриваются основные операторы и управляющие инструкции языка PowerShell.

В *восьмой главе* обсуждаются вопросы написания программных модулей (функций и сценариев) на встроенном в оболочку языке.

Девятая глава посвящена имеющимся в PowerShell средствам обработки ошибок и отладки сценариев.

Во второй части книги, *"Используем PowerShell"*, показаны примеры применения интерактивных команд и сценариев PowerShell для решения практических задач.

В *десятой главе* рассматриваются вопросы доступа из PowerShell к различным объектным моделям и технологиям, которые поддерживаются операци-

онной системой Windows (COM-объекты, подсистема Windows Management Instrumentations, служба каталогов ADSI, объекты платформы .NET). Приводятся примеры управления приложениями пакета Microsoft Office.

В *одиннадцатой главе* на примерах показывается, как с помощью команд PowerShell выполнять основные операции с объектами файловой системы.

В *двенадцатой главе* изучаются команды, позволяющие управлять процессами и службами.

В *тринадцатой главе* рассматриваются приемы работы из оболочки PowerShell с системным реестром Windows.

В *четырнадцатой главе* приводятся примеры работы с журналами событий Windows, использующие команды PowerShell.

В *пятнадцатой главе* обсуждаются задачи получения системной информации о настройках операционных систем и установленного программного обеспечения на локальном и удаленных компьютерах, а также команды для управления рабочими станциями в сети.

В *шестнадцатой главе* предложены команды PowerShell, которые можно использовать для составления HTML-отчетов об аппаратных компонентах компьютеров в сети.

В *семнадцатой главе* рассматриваются вопросы получения и настройки параметров сетевого протокола TCP/IP. Приводятся примеры сценариев на языке PowerShell, которые отправляют сообщения по электронной почте.

В *восемнадцатой главе* обсуждаются проблемы совместного использования различных средств автоматизации Windows (команд интерпретатора cmd.exe, сценариев WSH на языках VBScript или JScript, команд и сценариев PowerShell). Проводится сравнение языков cmd.exe и VBScript с языком PowerShell, приводятся примеры использования из PowerShell кода VBScript и JScript.

Кроме основного материала, в книге имеется два приложения. В *приложении 1* рассматривается объектная модель Windows Management Instrumentation (WMI), которая широко используется для выполнения различных задач администратора операционной системы. Предоставление доступа к этой модели из командной строки было одной из основных целей при разработке PowerShell.

В *приложении 2* описываются некоторые COM-объекты, которые могут оказаться полезными для администраторов и пользователей Windows, и даются примеры использования этих объектов из PowerShell.

Принятые в книге соглашения

Оболочка PowerShell — это интерактивная среда, поэтому во многих примерах показаны как команды, вводимые пользователем, так и ответ на них, генерируемый системой. Перед командой указывается приглашение PowerShell, обычно выглядящее как `PS C:\>`. Сама вводимая команда выделяется полужирным шрифтом, например **`Get-Process`**. В следующих нескольких строках приводится текст, возвращаемый системой в ответ на введенную команду, например:

```
PS C:\> Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  ProcessName
-------  ------    -----      ----- -----   ------     --  -----------
     99       5     1116        692    32     0.07    232 alg
     39       1      364        500    17     0.14   1636 ati2evxx
     57       3     1028       1408    30     0.38    376 atiptaxx
    412       6     2128       3600    26     6.50    808 csrss
     64       3      812       1484    29     0.19    316 ctfmon
    386      13    13748      14448    77    16.11   1848 explorer
    171       5     4512        584    44     0.20    428 GoogleT...
      0       0        0         16     0                0 Idle
    151       4     2908        992    41     1.05    412 kav
. . .
```

Многоточие здесь указывает на то, что для экономии места приведены не все строки, возвращаемые командой `Get-Process`.

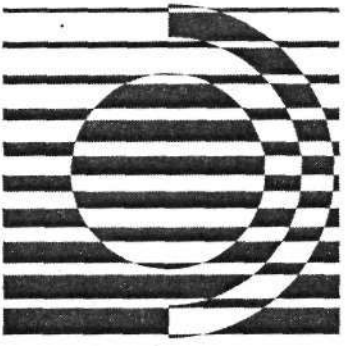
Иногда вводимые команды могут разбиваться на несколько строк. В этих случаях перед каждой дополнительной строкой команды указываются символы >>, например:

```
PS C:\> dir *.tmp | ForEach-Object {
>> $arr=$_.Name.split(".");
>> $newname=$arr[0]+".new";
>> ren $_.FullName $newname -PassThru;
>> }
>>
```

```
    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length   Name
----                 -------------     ------   ----
-a---         17.06.2008     11:39          0   3.new
. . .
```

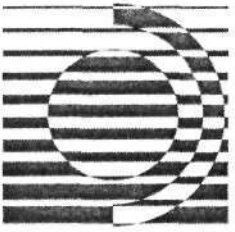
При описании операторов, функций и методов объектов используются стандартные соглашения. Названия подставляемых параметров и аргументов набраны курсивом, а необязательные параметры заключены в квадратные скобки `[]`, например:

```
CreateObject(strProgID [,strPrefix])
```



ЧАСТЬ I

Изучаем PowerShell



Глава 1

Windows PowerShell — результат развития технологий автоматизации

Прежде чем приступить непосредственно к изучению оболочки командной строки Windows PowerShell, попробуем ответить на ряд вопросов. Для чего, собственно, компании Microsoft потребовалось создавать этот совершенно новый инструмент и язык программирования? Какую пользу он может принести обычным пользователям и системным администраторам? Почему недостаточно было существующих средств?

Чтобы понять это, посмотрим, как в операционной системе Windows обстояло дело с *автоматизацией работы*, то есть решением различных задач в автоматическом режиме, без участия человека, до появления PowerShell.

Предшественники PowerShell в мире Windows

В настоящее время графический интерфейс Windows стал настолько привычным, что многие пользователи и начинающие администраторы даже не задумываются об альтернативных способах управления операционной системой с помощью *командной строки* (command line) и различных *сценариев* (scripts). Зачастую они просто не знают о тех преимуществах, которые дают эти инструменты с точки зрения автоматизации работы.

Подобная ситуация обусловлена тем, что исторически командная строка всегда была слабым местом операционной системы Windows (по сравнению с UNIX-системами). Причиной этого, прежде всего, является то, что компания Microsoft изначально ориентировалась на широкую аудиторию неискушенных пользователей, не желающих особо вникать в технические детали выполнения тех или иных действий в системе. Поэтому основные усилия разработчиков операционной системы направлялись на улучшение графической оболочки для более комфортной работы непрофессионалов, а не на создание рабочей среды для специалистов или опытных пользователей.

Как показало время, с коммерческой точки зрения на рынке персональных (домашних или офисных) компьютеров эта стратегия оказалась более чем успешной: миллионы людей используют графический интерфейс Windows для запуска нужных им программ, работы в офисных пакетах, просмотра фильмов и т. п. Да и управлять одним Windows-сервером сегодня несложно: операционная система предлагает удобные графические средства для настройки различных параметров и выполнения ежедневных администраторских задач, а с помощью службы терминалов легко можно работать на удаленном сервере, физически расположенном хоть на другом континенте.

Однако подобная модель управления не является масштабируемой: если с помощью стандартных графических инструментов администрировать не один, а десять серверов, то последовательность изменений настроек в диалоговых окнах придется повторить десять раз. Следовательно, в этом случае остро встает вопрос об автоматизации выполнения рутинных операций (например, проведения инвентаризации оборудования и программного обеспечения, мониторинга работы служб, анализа журналов событий и т. д.) на множестве компьютеров. Помочь в этом могут либо специальные (как правило, тяжеловесные и недешевые) приложения типа Microsoft Systems Management Server (SMS), либо сценарии, которые пишутся администраторами самостоятельно (на языке оболочки командной строки или на специальных языках сценариев) и поддерживаются непосредственно операционной системой, без установки сторонних программных продуктов.

Поэтому для профессионала, занимающегося администрированием информационных систем на базе Windows, знание возможностей командной строки, сценариев и технологий автоматизации, поддерживаемых данной операционной системой, просто необходимо.

При этом, однако, неправильно было бы думать, что командная строка или сценарии нужны только администраторам. Ведь рутинные ежедневные задачи пользователей (связанные, например, с копированием или архивированием файлов, подключением или отключением сетевых ресурсов и т. п.), которые обычно выполняются с помощью графического интерфейса проводника Windows, можно полностью самостоятельно автоматизировать, написав нехитрый командный файл, состоящий всего из нескольких строчек! Однако для человека, не знающего основные команды Windows и такие базовые возможности операционной системы, как перенаправление ввода/вывода и конвейеризация команд, некоторые простейшие задачи могут показаться нетривиальными. Попробуйте, например, пользуясь только графическими средствами, сформировать файл, содержащий имена файлов из всех подкаталогов какого-либо каталога! А ведь для этого достаточно выполнить единст-

венную команду `dir` (с определенными ключами) и перенаправить вывод этой команды в нужный текстовый файл. Например, следующая команда создаст текстовый файл c:\list_mp3.txt, в котором будут записаны имена всех файлов с расширением mp3, находящихся в каталоге c:\music или в каком-либо его подкаталоге:

```
dir /s /b c:\music\*.mp3 > c:\list_mp3.txt
```

Задумаемся теперь, каким же нам хотелось бы видеть инструмент для автоматизации работы в операционной системе, какими возможностями он должен обладать? Желательно, чтобы в нем было реализовано следующее:

- работа в разных версиях операционной системы (в идеальном случае во всех) без установки какого-либо дополнительного программного обеспечения;
- интеграция с командной строкой (непосредственное выполнение вводимых с клавиатуры команд);
- согласованный и непротиворечивый синтаксис команд и утилит;
- наличие подробной встроенной справки по командам с примерами использования;
- возможность выполнения сценариев, составленных на простом для изучения языке;
- возможность использования всех технологий, поддерживаемых операционной системой.

В UNIX-системах в качестве инструмента автоматизации выступает стандартная оболочка (sh) или ее модификации (bash, ksh, csh и т. д.), причем этот аспект операционной системы стандартизирован в рамках POSIX (стандарт мобильных систем).

В операционной системе Windows дело обстоит сложнее. На сегодняшний день одного "идеального" средства автоматизации, удовлетворяющего сразу всем перечисленным выше требованиям, в Windows нет. В последних версиях операционной системы одновременно поддерживаются несколько стандартных инструментов автоматизации, сильно отличающихся друг от друга: оболочка командной строки cmd.exe, среда выполнения сценариев Windows Script Host (WSH), оболочка WMI Command-line (WMIC) и, наконец, новинка и предмет нашего изучения — оболочка Microsoft PowerShell. Поэтому администратору или пользователю Windows приходится выбирать, каким именно подходом воспользоваться для решения определенной задачи, а для этого желательно иметь четкое представление о сильных и слабых сторонах всех доступных средств автоматизации.

Оболочка командной строки command.com/cmd.exe

Во всех версиях операционной системы Windows поддерживается интерактивная *оболочка командной строки* (command shell), и по умолчанию устанавливается определенный набор утилит командной строки (количество и состав этих утилит зависит от версии операционной системы). Вообще, любую операционную систему можно представить в виде совокупности *ядра системы*, которое имеет доступ к аппаратуре и управляет файлами и процессами, и *оболочки* (*командного интерпретатора*) с утилитами, которые позволяют пользователю получить доступ к функциональности ядра операционной системы. Механизм работы оболочек в разных системах одинаков: в ответ на приглашение ("подсказку", prompt), выдаваемое находящейся в ожидании оболочкой, пользователь вводит некоторую команду (функциональность этой команды может быть реализована либо самой оболочкой, либо определенной внешней утилитой), оболочка выполняет ее, при необходимости выводя на экран какую-либо информацию, после чего снова выводит приглашение и ожидает ввода следующей команды (рис. 1.1).

Замечание

С технической точки зрения оболочка представляет собой построчный интерпретатор простого языка сентенциального (директивного) программирования, в качестве операторов которого могут использоваться исполняемые программы.

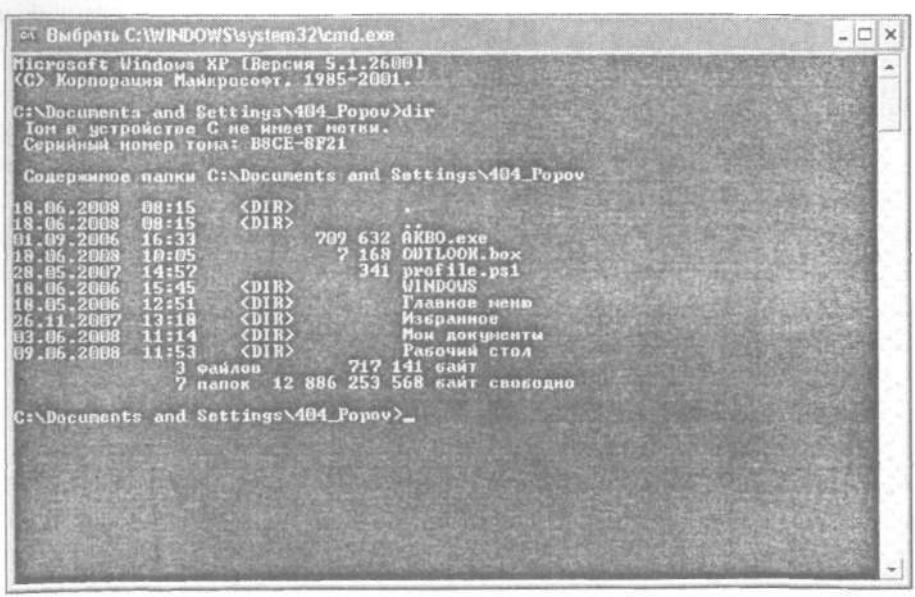


Рис. 1.1. Результат выполнения команды dir в оболочке cmd.exe

Наряду с интерактивным режимом работы оболочки, как правило, поддерживают и пакетный режим, в котором система последовательно выполняет команды, записанные в текстовом файле-сценарии. Оболочка Windows не является исключением, с точки зрения программирования язык командных файлов Windows может быть охарактеризован следующим образом:

- реализация сентенциальной (директивной) парадигмы программирования;
- выполнение в режиме построчной интерпретации;
- наличие управляющих конструкций;
- поддержка нескольких видов циклов (в том числе специальных циклов для обработки текстовых файлов);
- наличие оператора присваивания (установки значения переменной);
- возможность использования внешних программ (команд) операционной системы в качестве операторов и обработки их кодов возврата;
- наличие нетипизированных переменных, которые декларируются первым упоминанием (значение переменных могут интерпретироваться как числа и использоваться в выражениях целочисленной арифметики).

Начиная с версии Windows NT, оболочка командной строки представляется интерпретатором cmd.exe, который расширяет возможности оболочки command.com операционной системы MS-DOS. В свою очередь функциональность командного интерпретатора command.com была позаимствована из операционной системы CP/M, командный интерпретатор которой представлял собой значительно упрощенный и урезанный вариант оболочки UNIX-систем.

Таким образом, оболочка командной строки MS-DOS изначально уступала UNIX-оболочкам по удобству работы и развитости языка сценариев. В командной оболочке Windows NT (cmd.exe), несмотря на все сделанные улучшения, не удалось преодолеть данное отставание ни в режиме интерактивной работы (например, в cmd.exe отсутствует поддержка псевдонимов для длинных названий команд и не реализовано автоматическое завершение команд при вводе их с клавиатуры), ни в синтаксисе или возможностях языка командных файлов.

Ситуация усугублялась тем, что Windows всегда проигрывала UNIX-системам в количестве и функциональных возможностях стандартных (не требующих дополнительной установки) утилит командной строки, а также в качестве и полноте встроенной справочной системы по командам оболочки.

> ***Замечание***
>
> Для того чтобы прочитать встроенную справку для определенной команды, нужно вызвать эту команду с ключом `/?` (например, `xcopy /?`). Общий справочник по командной строке находится в файле %WinDir%\Help\Ntcmds.chm.

На практике проблему отсутствия нужной функциональности у стандартных команд приходится решать либо с помощью утилит пакета Windows Resource Kit для соответствующей версии операционной системы, либо путем поиска подходящей утилиты сторонних производителей. Кроме того, в Windows можно пользоваться POSIX-совместимыми утилитами и оболочками с помощью пакета Microsoft Services For UNIX (SFU). Данный продукт разрабатывался еще для Windows NT и первоначально не входил в состав операционной системы, его нужно было приобретать за отдельную плату. В дальнейшем пакет SFU стал бесплатным и даже был включен в состав операционной системы Windows Server 2003 R2.

Итак, учитывая все сказанное ранее, мы можем сделать следующий вывод: оболочка командной строки cmd.exe и командные файлы — наиболее универсальные и простые в изучении средства автоматизации работы в Windows, доступные во всех версиях операционной системы. При этом, однако, оболочка cmd.exe и командные файлы существенно проигрывают аналогичным инструментам в UNIX-системах и не обеспечивают доступ к объектным моделям, поддерживаемым операционной системой (COM, WMI, .NET).

Сервер сценариев Windows Script Host (WSH)

Следующим шагом в развитии средств и технологий автоматизации в операционной системе Windows стало появление сервера сценариев Windows Script Host (WSH). Этот инструмент разработан для всех версий Windows и позволяет непосредственно в операционной системе выполнять сценарии на полноценных языках сценариев (по умолчанию, VBScript и JScript), которые до этого были доступны только внутри HTML-страниц и работали в контексте безопасности веб-браузера (в силу этого подобные сценарии, например, могли не иметь доступа к файловой системе локального компьютера).

По сравнению с командными файлами интерпретатора cmd.exe сценарии WSH имеют несколько преимуществ.

Во-первых, VBScript и JScript — это полноценные алгоритмические языки, имеющие встроенные функции и методы для обработки символьных строк, выполнения математических операций, обработки исключительных ситуаций и т. д.; кроме того, для написания сценариев WSH может использоваться любой другой язык сценариев (например, широко распространенный в UNIX-системах Perl), для которого установлен соответствующий модуль поддержки.

Во-вторых, WSH поддерживает несколько собственных объектов, свойства и методы которых позволяют решать некоторые часто возникающие повседневные задачи администратора операционной системы: работа с сетевыми

ресурсами, переменными среды, системным реестром, ярлыками и специальными папками Windows, запуск и управление работой других приложений. Например, в сценарии MakeShortcut.vbs с помощью объекта WshShell создается ярлык на сетевой ресурс: сайт компании Microsoft (листинг 1.1).

Листинг 1.1. Создание ярлыка из сценария (VBScript)

```
'*****************************************************************
'* Имя: MakeShortcut.vbs
'* Язык: VBScript
'* Описание: Создание ярлыков из сценария
'*****************************************************************
Dim WshShell,oUrlLink
' Создаем объект WshShell
Set WshShell=WScript.CreateObject("WScript.Shell")
' Создаем ярлык на сетевой ресурс
Set oUrlLink = WshShell.CreateShortcut("Microsoft Web Site.URL")
' Устанавливаем URL
oUrlLink.TargetPath = "http://www.microsoft.com"
' Сохраняем ярлык
oUrlLink.Save
'************* Конец ***********************************************
```

В-третьих, из сценариев WSH можно обращаться к службам любых приложений-серверов автоматизации, которые регистрируют в операционной системе свои объекты (скажем, программ из пакета Microsoft Office). Например, в сценарии PrintInWord.vbs происходит подключение к серверу автоматизации Microsoft Word и вывод строк текста в окно этого приложения (листинг 1.2).

Листинг 1.2. Использование сервера автоматизации Microsoft Word (VBScript)

```
'*****************************************************************
' Имя: PrintInWord.vbs
' Язык: VBScript
' Описание: Использование из сценария внешнего объекта
'           автоматизации (Microsoft Word)
'*****************************************************************
Option Explicit

Dim WA,WD,Sel   ' Объявляем переменные
```

```
'Создаем объект-приложение Microsoft Word
Set WA=WScript.CreateObject("Word.Application")
' Можно было использовать конструкцию
' Set WA=CreateObject("Word.Application")

Set WD=WA.Documents.Add  'Создаем новый документ (объект Document)
WA.Visible=true  ' Делаем Word видимым
Set Sel=WA.Selection  'Создаем объект Selection
Sel.Font.Size=14 'Устанавливаем размер шрифта
Sel.ParagraphFormat.Alignment=1  'Выравнивание по центру
Sel.Font.Bold=true  'Устанавливаем полужирный шрифт
Sel.TypeText "Привет!" & vbCrLf  'Печатаем строку текста
Sel.Font.Bold=false  'Отменяем полужирный шрифт
Sel.ParagraphFormat.Alignment=0  'Выравнивание по левому краю
'Печатаем строку текста
Sel.TypeText "Эти строки напечатаны с помощью WSH."
WD.PrintOut   'Выводим документ на принтер
'*************  Конец ********************************************
```

Наконец, сценарии WSH позволяют работать с объектами информационной модели Windows Management Instrumentation (WMI), обеспечивающей программный интерфейс управления всеми компонентами операционной модели, а также с объектами службы каталогов Active Directory Service Interface (ADSI) (объектная модель WMI подробно обсуждается в *приложении 1*).

Следует также отметить, что технология WSH поддерживается в Windows уже довольно давно, в Интернете (в том числе на сайте Microsoft) можно найти множество готовых сценариев, выполняющих ту или иную операцию, и при определенных навыках и знаниях быстро "подогнать" эти сценарии под свои конкретные задачи.

Поговорим теперь о слабых местах WSH. Прежде всего, сам по себе WSH — это только среда выполнения сценариев, а не оболочка; WSH не интегрирован с командной строкой, то есть отсутствует режим, в котором можно было вводить команды с клавиатуры и сразу видеть результат их выполнения.

Большим минусом для WSH является то, что в операционной системе по умолчанию нет полноценной подробной справочной информации по объектам WSH и языкам VBScript/JScript (документацию приходится искать в Интернете на сайте Microsoft). Другими словами, если вы, например, не помните синтаксис определенной команды VBScript/JScript или точное название свойства объекта WSH, под рукой у вас нет распечатанной документации,

а компьютер не имеет выхода в Интернет, то написать корректный сценарий вам просто не удастся. (В данном аспекте командные файлы более универсальны, так как практически у всех команд есть, по крайней мере, встроенное описание используемых ими ключей, а в операционной системе имеется справочный файл с информацией обо всех стандартных командах.)

Наконец, сценарии WSH представляют собой довольно серьезную потенциальную угрозу с точки зрения безопасности, известно большое количество вирусов, использующих WSH для выполнения деструктивных действий.

Таким образом, можно дать следующую общую оценку: сценарии WSH — это универсальный инструмент, который в любой версии операционной системы Windows позволяет решать задачи автоматизации практически любой степени сложности, но требует при этом большой работы по изучению самих языков сценариев и ряда смежных технологий управления операционной системой (WMI, ADSI и т. п.).

Оболочка WMI Command-line (WMIC)

Как уже упоминалось, в операционной системе Windows поддерживается информационная модель Windows Management Instrumentation (WMI), которая занимает важное место среди технологий, инструментов и средств автоматизации. В основе данной технологии лежит схема CIM (Common Information Model), которая представляет физическую и логическую структуры компьютерной системы в виде единой расширяемой объектно-ориентированной информационной модели и определяет единые интерфейсы для получения информации о любом компоненте этой модели.

Изначально работать с WMI можно было либо с помощью специальных графических утилит, либо путем составления довольно сложных сценариев WSH. В состав операционных систем Windows XP и Windows Server 2003 была включена утилита WMIC (WMI Command-line), позволяющая обращаться к подсистеме WMI непосредственно из командной строки. Оболочка WMIC поддерживает навигацию по информационной схеме WMI локального или удаленного компьютера, позволяя выполнять WQL-запросы к классам и объектам WMI. При этом вместо сложных названий классов WMI используются простые псевдонимы, причем можно создавать собственные псевдонимы, что делает информационную схему WMIC расширяемой. Например, классу `Win32_OperatingSystem` соответствует псевдоним `os`. Если набрать в командной строке WMIC команду `os` и нажать <Enter>, то мы увидим на экране свойства операционной системы, установленной на компьютере (рис. 1.2).

По умолчанию WMIC поддерживает около 80 псевдонимов, с помощью которых можно выполнить полторы сотни методов и получить значения множества свойств. Важной особенностью WMIC является то, что вывод команд

может быть организован в различные форматы: на экран, в текстовый файл, в XML- и HTML-документы, в MOF-файл, в текстовый файл с разделителями или в любой другой формат, определяемый пользователем с помощью таблиц стилей XSL (eXtensible Stylesheet Language).

Рис. 1.2. Результат выполнения команды OS в оболочке WMIC

Одна команда WMIC может быть применена сразу к нескольким удаленным компьютерам с любой 32-разрядной версией Windows, при этом наличие WMIC на удаленной машине не требуется, необходима только установка ядра WMI и соответствующая настройка прав доступа к WMI. Кроме этого, команды WMI могут использоваться в пакетных файлах Windows, что позволяет простыми средствами автоматизировать работу с WMI на локальных или удаленных компьютерах.

В качестве недостатка WMIC можно отметить отсутствие встроенной полноценной поддержки и обработки событий WMI. Как показало время, оболочка WMIC оказалась не особенно удачной, так как в этом продукте акцент был сделан на функциональные особенности WMI, а не на удобстве работы пользователя.

Причины и цели создания оболочки PowerShell

Итак, к началу XXI века в операционной системе Windows поддерживались три разных инструмента для автоматизации работы: оболочки командной

строки cmd.exe и WMIC, а также сервер сценариев WSH. Зачем же компании Microsoft понадобилась разработка еще одной совершенно новой оболочки командной строки со своим языком сценариев?

Дело в том, что у каждого из перечисленных инструментов автоматизации имелись довольно серьезные недостатки, не позволявшие сказать, что Windows обладает по-настоящему мощным и эффективным средством для работы с командной строкой и написания сценариев (см. табл. 1.1). С одной стороны, функциональности и гибкости языка оболочки cmd.exe было явно недостаточно, а с другой стороны, сценарии WSH, работающие с объектными моделями ADSI и WMI, оказались слишком сложными для пользователей среднего уровня и начинающих администраторов.

Таблица 1.1. *Требования к инструменту автоматизации*

Требование	cmd.exe	WSH	WMIC
Работа во всех версиях операционной системы без установки дополнительного программного обеспечения	Да	Да	Нет (только Windows XP и выше)
Интеграция с командной строкой	Да	Нет	Да
Согласованный и непротиворечивый синтаксис команд и утилит	*Нет*	*Нет*	Да
Поддержка *псевдонимов* (кратких синонимов) для длинных названий команд	Нет	Нет	Да
Автоматическое завершение команд и имен файлов при вводе их с клавиатуры	Частично (автоматическое завершение имен файлов и папок)	Нет	Нет
Поддержка *истории введенных команд* с возможностью их повторного вызова, просмотра и редактирования	Да	Нет	Да
Наличие подробной встроенной справки по командам с примерами использования	Частично	Нет	Да

Таблица 1.1 (окончание)

Требование	cmd.exe	WSH	WMIC
Возможность автоматического выполнения сценариев	Да (язык командных файлов)	Да (языки сценариев VBScript, JScript и т. д.)	Частично (команды WMIC можно встраивать в командные файлы)
Доступ и использование всех технологий и возможностей, поддерживаемых операционной системой	Нет (нет прямого доступа к объектам COM, WMI, ADSI, .NET)	Да	Нет (доступ только к объектам WMI)

Начав дорабатывать WMIC, специалисты Microsoft поняли, что можно реализовать оболочку, которая не ограничивалась бы только работой с объектами WMI, а также предоставляла бы доступ к любым классам платформы .NET Framework, обеспечивая тем самым возможность пользоваться из командной строки всеми мощными функциональными возможностями данной среды.

Новая оболочка *Windows PowerShell* (первоначально она называлась Monad) была задумана разработчиками Microsoft как более мощная среда для написания сценариев и работы из командной строки. Разработчики PowerShell преследовали несколько целей. Главная и наиболее амбициозная из них — создать среду составления сценариев, которая наилучшим образом подходила бы для современных версий операционной системы Windows и была бы более функциональной, расширяемой и простой в использовании, чем какой-либо аналогичный продукт для любой другой операционной системы. В первую очередь эта среда должна была подходить для решения задач, стоящих перед системными администраторами (тем самым Windows получила бы дополнительное преимущество в борьбе за сектор корпоративных платформ), а также удовлетворять требованиям разработчиков программного обеспечения, предоставляя им средства для быстрой реализации интерфейсов управления создаваемыми приложениями.

Для достижения этих целей были решены следующие задачи:

- **Обеспечение прямого доступа из командной строки к объектам COM, WMI и .NET.** В новой оболочке присутствуют команды, позволяющие в интерактивном режиме работать с COM-объектами, а также с экземплярами классов, определенных в информационных схемах WMI и .NET.
- **Организация работы с произвольными источниками данных в командной строке по принципу файловой системы.** Например, навигация по системному реестру или хранилищу цифровых сертификатов выполняется

из командной строки с помощью аналога команды `cd` интерпретатора cmd.exe.

- **Разработка интуитивно понятной унифицированной структуры встроенных команд, основанной на их функциональном назначении.** В новой оболочке имена всех внутренних команд (в PowerShell они называются *командлетами*) соответствуют шаблону "глагол-существительное", например, `Get-Process` (получить информацию о процессе), `Stop-Service` (остановить службу), `Clear-Host` (очистить экран консоли) и т. д. Для одинаковых параметров внутренних команд используются стандартные имена, структура параметров во всех командах идентична, все команды обрабатываются одним синтаксическим анализатором. В результате облегчается изучение и запоминание команд.
- **Обеспечение возможности расширения встроенного набора команд.** Внутренние команды PowerShell могут дополняться командами, создаваемыми пользователем. При этом они полностью интегрируются в оболочку, информация о них может быть получена из стандартной справочной системы PowerShell.
- **Организация поддержки знакомых команд из других оболочек.** В PowerShell на уровне псевдонимов собственных внутренних команд поддерживаются наиболее часто используемые стандартные команды из оболочки cmd.exe и UNIX-оболочек. Например, если пользователь, привыкший работать с UNIX-оболочкой, выполнит `ls`, то он получит ожидаемый результат: список файлов в текущем каталоге (то же самое относится к команде `dir`).
- **Разработка полноценной встроенной справочной системы для внутренних команд.** Для большинства внутренних команд в справочной системе дано подробное описание и примеры использования. В любом случае встроенная справка по любой внутренней команде будет содержать краткое описание всех ее параметров.
- **Реализация автоматического завершения при вводе с клавиатуры имен команд, их параметров, а также имен файлов и папок.** Данная возможность значительно упрощает и ускоряет ввод команд с клавиатуры.

Разработчики старались собрать в PowerShell все лучшие аспекты других оболочек командной строки из разных операционных систем. По их словам, сильное влияние на PowerShell оказали следующие продукты:

- bash, ksh (конвейеризация или композиция команд);
- AS/400, VMS (стандартные названия команд, ускоряющие изучение);
- Tcl, WSH (поддержка встраиваемости и нескольких языков);
- Perl, Python (выразительность и стиль).

Отметим, что PowerShell одновременно является и оболочкой командной строки (пользователь может работать в интерактивном режиме) и средой выполнения сценариев, которые пишутся на специальном языке PowerShell.

Интерактивный сеанс в PowerShell похож на работу в оболочке UNIX-систем. Все команды в PowerShell имеют подробную встроенную справку (для большинства команд приводятся примеры их использования), поддерживается функция автоматического завершения названий команд и их параметров при вводе с клавиатуры, для многих команд имеются псевдонимы, аналогичные названиям UNIX-утилит (`ls`, `pwd`, `tee` и т. д.).

Отдельное внимание было уделено вопросам безопасности при работе со сценариями (например, запустить сценарий можно только с указанием полного пути к нему, а по умолчанию запуск сценариев PowerShell в системе вообще запрещен).

Язык PowerShell несложен для изучения, писать на нем сценарии, обращающиеся к внешним объектам, проще, чем на VBScript или JScript. В целом, оболочка PowerShell намного удобнее и мощнее своих предшественников (cmd.exe и WSH), а основным недостатком, сдерживающим распространение нового инструмента, является тот факт, что PowerShell работает не во всех версиях операционной системы Windows. Оболочкой можно пользоваться только на версиях не ниже Windows XP Service Pack 2 с установленным пакетом .NET Framework 2.0.

Главной особенностью среды PowerShell, отличающей ее от всех других оболочек командной строки, является то, что единицей обработки и передачи информации здесь является объект, а не строка текста. В командной строке PowerShell вывод результатов команды представляет собой не текст (в смысле последовательности байтов), а объект (данные вместе со свойственными им методами). В силу этого работать в PowerShell становится проще, чем в традиционных оболочках, так как не нужно выполнять никаких манипуляций по выделению нужной информации из символьного потока.

Отличие PowerShell от других оболочек — ориентация на объекты

При разработке любого языка программирования одним из основных является вопрос о том, какие типы данных и каким образом будут в нем представлены. При создании PowerShell разработчики решили не изобретать ничего нового и воспользоваться унифицированной объектной моделью .NET. Данный выбор был сделан по нескольким причинам.

Во-первых, *платформа .NET* повсеместно используется при разработке программного обеспечения для Windows и предоставляет, в частности, общую информационную схему, с помощью которой разные компоненты операционной системы могут обмениваться данными друг с другом.

Во-вторых, объектная модель .NET является *самодокументируемой*: каждый объект .NET содержит информацию о своей структуре. При интерактивной работе это очень полезно, так как появляется возможность непосредственно из командной строки выполнить запрос к определенному объекту и увидеть описание его свойств и методов, то есть понять, какие именно манипуляции можно проделать с данным объектом, не изучая дополнительной документации с его описанием.

В-третьих, работая в оболочке с объектами, можно с помощью их свойств и методов легко получать нужные данные, не занимаясь разбором и анализом символьной информации, как это происходит во всех традиционных оболочках командной строки, ориентированных на текст. Рассмотрим пример. В Windows XP есть консольная утилита tasklist.exe, которая выдает информацию о процессах, запущенных в системе:

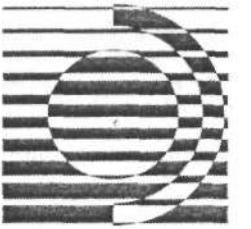
```
C:\> tasklist

Имя образа                   PID Имя сессии        № сеанса       Память
========================= ====== ================ ======== ============
System Idle Process            0                         0         16 КБ
System                         4                         0         32 КБ
smss.exe                     560                         0         68 КБ
csrss.exe                    628                         0      4 336 КБ
winlogon.exe                 652                         0      3 780 КБ
services.exe                 696                         0      1 380 КБ
lsass.exe                    708                         0      1 696 КБ
svchost.exe                  876                         0      1 164 КБ
svchost.exe                  944                         0      1 260 КБ
svchost.exe                 1040                         0     10 144 КБ
svchost.exe                 1076                         0        744 КБ
svchost.exe                 1204                         0        800 КБ
spoolsv.exe                 1296                         0      1 996 КБ
kavsvc.exe                  1516                         0      9 952 КБ
klnagent.exe                1660                         0      5 304 КБ
klswd.exe                   1684                         0         64 КБ
```

Предположим, что мы в командном файле интерпретатора `cmd.exe` с помощью этой утилиты хотим определить, сколько оперативной памяти тратит процесс kavsvc.exe. Для этого нужно выделить из выходного потока команды `tasklist` соответствующую строку, извлечь из нее подстроку, содержащую нужное число, и убрать пробелы между разрядами (при этом следует учесть, что в зависимости от настроек операционной системы разделителем разрядов может быть не пробел, а другой символ). В PowerShell аналогичная задача решается с помощью команды `Get-Process`, которая возвращает коллекцию объектов, каждый из которых соответствует одному запущенному процессу. Для определения памяти, затрачиваемой процессом kavsvc.exe, нет необходимости в дополнительных манипуляциях с текстом, достаточно просто взять значение свойства `WS` объекта, соответствующего данному процессу.

Наконец, объектная модель .NET позволяет PowerShell напрямую использовать функциональность различных библиотек, являющихся частью платформы .NET. Например, чтобы узнать, каким днем недели было 9 ноября 1974 года, в PowerShell можно выполнить следующую команду:

```
(Get-Date "09.11.1974").DayOfWeek
```

В этом случае команда `Get-Date` возвращает .NET-объект `DateTime`, имеющий свойство, при обращении к которому вычисляется день недели для соответствующей даты. Таким образом, разработчикам PowerShell не нужно создавать специальную библиотеку для работы с датами и временем — они просто берут готовое решение из платформы .NET.



Глава 2

Первые шаги в PowerShell. Основные понятия

Итак, приступим к работе в новой оболочке командной строки Windows от Microsoft! Если у вас загружена операционная система Windows Vista, то оболочка PowerShell уже установлена. Если же вы работаете с другой версией Windows, то PowerShell нужно предварительно загрузить и установить.

Загрузка и установка PowerShell

Оболочка PowerShell может работать не во всех версиях операционной системы Windows, ее можно использовать в Windows XP SP2, Windows Server 2003 SP1 и более поздних версиях. Кроме этого, в системе должна быть установлена платформа .NET Framework 2.0 (в Windows XP данную среду придется устанавливать дополнительно, дистрибутив можно загрузить с сайта Microsoft по адресу **http://msdn.microsoft.com/netframework/downloads/updates/default.aspx**).

Загрузить PowerShell можно с сайта Microsoft, открыв страницу **http://microsoft.com/powershell**, где имеются ссылки на файл установки последней версии и пакеты документации на разных языках. Запустив загруженный файл, следуйте указаниям мастера установки.

В 32-разрядных версиях Windows оболочка PowerShell устанавливается по умолчанию в каталог %SystemRoot%\System32\WindowsPowerShell\v1.0. В 64-разрядных версиях Windows 32-разрядная версия PowerShell устанавливается в каталог %SystemRoot%\SystemWow64\WindowsPowerShell\v1.0, а 64-разрядная версия Windows PowerShell устанавливается в каталог %SystemRoot%\System32\WindowsPowerShell\v1.0.

Запуск оболочки

Установив оболочку в системе, можно начать новый интерактивный сеанс. Для этого следует нажать кнопку **Пуск**, открыть меню **Все программы** и выбрать элемент **Windows PowerShell**. Другой вариант запуска оболочки — выбрать пункт **Выполнить...** в меню **Пуск**, ввести имя файла `powershell` и нажать кнопку **OK**.

В результате откроется новое командное окно с приглашением вводить команды (рис. 2.1).

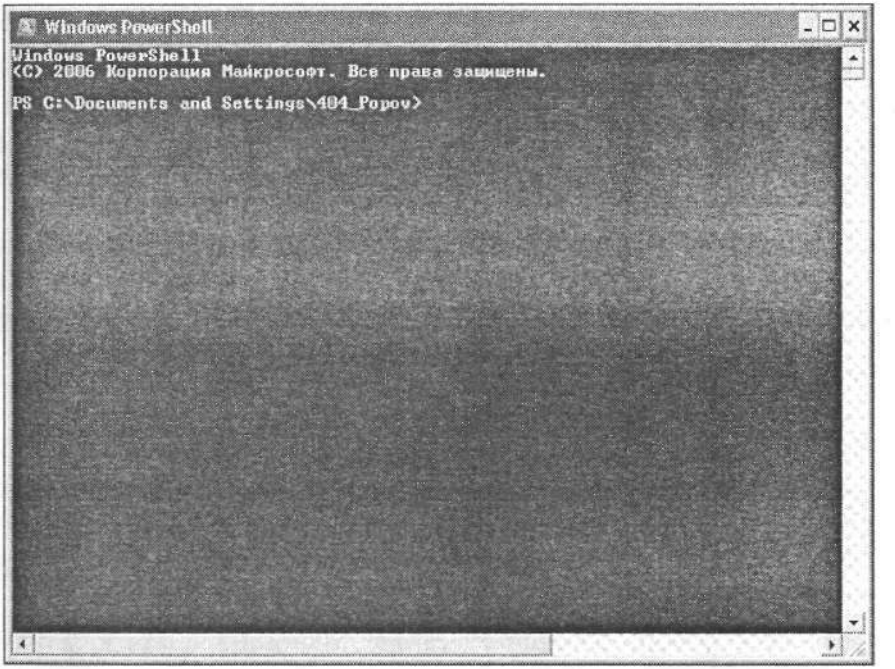


Рис. 2.1. Окно командной оболочки PowerShell

Работают ли знакомые команды?

Выполним первую команду в PowerShell. Пусть это будет что-то знакомое по работе с оболочкой cmd.exe, например, команда `dir`. Вспомним сначала, как выглядит вывод этой команды в cmd.exe:

```
C:\Documents and Settings\User> dir
 Том в устройстве C имеет метку C
```

```
 Серийный номер тома: 44D2-29CE

 Содержимое папки C:\Documents and Settings\User

22.01.2008  22:10    <DIR>          .
22.01.2008  22:10    <DIR>          ..
12.05.2005  16:16    <DIR>          DoctorWeb
24.05.2006  20:22             8 304 gsview32.ini
06.11.2004  13:03    <DIR>          Phone Browser
04.10.2004  14:33    <DIR>          Главное меню
05.12.2007  00:49    <DIR>          Избранное
31.10.2007  21:03    <DIR>          Мои документы
23.01.2008  11:13    <DIR>          Рабочий стол
               1 файлов          8 304 байт
               8 папок   1 734 508 544 байт свободно
```

Итак, на экран выводится список файлов и подкаталогов в текущем каталоге, а также некоторая дополнительная информация.

Теперь запустим `dir` в PowerShell (отметим, что в новой оболочке команды по-прежнему обрабатываются без учета регистра):

```
PS C:\Documents and Settings\User> dir

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\Documents and Set-
tings\User

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d---s          14.12.2007     10:10            Cookies
d----          12.05.2005     17:16            DoctorWeb
d----          06.11.2004     13:03            Phone Browser
d---s          22.09.2004     23:49            UserData
d-r--          04.10.2004     15:33            Главное меню
d-r--          05.12.2007      0:49            Избранное
d-r--          31.10.2007     21:03            Мои документы
d----          05.12.2007     10:54            Рабочий стол
-a---          24.05.2006     21:22       8304 gsview32.ini
```

Как и следовало ожидать, на экран также выводится список файлов и подкаталогов в текущем каталоге, хотя внешний вид выходной информации команды `dir` в оболочках cmd.exe и PowerShell различается.

Работая в оболочке cmd.exe, мы с помощью символа > (знак "больше") могли выводить результат выполнения команд не на экран, а в текстовый файл

(эта процедура называется *перенаправлением вывода*). Попробуем сделать то же самое в PowerShell:

```
PS C:\Documents and Settings\User> dir > c:\dir.txt
```

Данная команда выполнилась без каких-либо сообщений на экране. Выведем теперь на экран содержимое файла c:\dir.txt, воспользовавшись для этого командой `type`:

```
PS C:\Documents and Settings\User> type c:\dir.txt


    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\Documents and Set-
tings\User


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d---s         01.02.2008     19:05            Cookies
d----         12.05.2005     17:16            DoctorWeb
d----         06.11.2004     13:03            Phone Browser
d---s         22.09.2004     23:49            UserData
d-r--         04.10.2004     15:33            Главное меню
d-r--         05.12.2007      0:49            Избранное
d-r--         31.10.2007     21:03            Мои документы
d----         23.01.2008     11:13            Рабочий стол
-a---         24.05.2006     21:22       8304 gsview32.ini
```

Замечание

Если файл, в который с помощью символа > перенаправляется вывод, уже существует, то его содержимое будет перезаписано. В оболочке PowerShell также поддерживается перенаправление вывода в режиме добавления информации в конец существующего файла — для этого используются символы >>. Перенаправление ввода с помощью символа < в PowerShell не поддерживается.

Итак, начало положено: мы выполнили в новой оболочке несколько знакомых команд и убедились, что PowerShell поддерживает перенаправление выходной информации в текстовый файл.

Вычисление выражений

Кроме выполнения команд, в PowerShell можно вычислять выражения, то есть пользоваться оболочкой как калькулятором (в оболочке cmd.exe эта возможность отсутствовала). Например:

```
PS C:\> 2+3
5
```

```
PS C:\> 10-2*3
4
PS C:\> (10-2)*3
24
```

Как видите, результат вычислений сразу же выводится на экран, и нет необходимости использовать для этого какую-либо специальную команду (эту особенность PowerShell следует запомнить на будущее).

Предыдущие примеры показывают, что PowerShell справляется с вычислением целочисленных выражений, в том числе со скобками. Проверим выражение, результатом которого является число с плавающей точкой:

```
PS C:\> 10/3
3.33333333333333
```

Здесь тоже все в порядке, результат вычислен верно. На самом деле в PowerShell можно выполнять и более сложные вычисления, включающие в себя различные математические функции. Для этого используются методы .NET-класса `System.Math`. Например, следующая команда вычисляет и выводит на экран корень квадратный из числа 169:

```
PS C:\> [System.Math]::Sqrt(169)
13
```

> ***ЗАМЕЧАНИЕ***
>
> Более подробно использование методов .NET-классов будет обсуждаться в *главе 10*.

Результат вычисления выражения в PowerShell можно сохранять в переменной и пользоваться этой переменной в других выражениях. Например:

```
PS C:\> $a=10/2
PS C:\> $a
5
PS C:\> $a*3
15
```

Отметим здесь, что в PowerShell имена переменных должны начинаться со знака доллара ($). Более подробно вопросы, связанные с переменными, будут обсуждаться в *главе 6*.

Типы команд PowerShell

Вспомним сначала, какие типы команд имелись в оболочке cmd.exe. Здесь все команды разделялись на *внутренние*, которые распознавались и выполнялись непосредственно интерпретатором cmd.exe, и *внешние*, которые представляли

собой отдельные исполняемые модули. К внутренним относятся, например, команды `dir` и `copy`, а к внешним — `xcopy` и `more`. Кроме этого, напомним, оболочка cmd.exe поддерживала сценарии — командные файлы.

В оболочке PowerShell поддерживаются команды четырех типов: командлеты, функции, сценарии и внешние исполняемые файлы.

Командлеты

Первый тип команд PowerShell — так называемые *командлеты* (cmdlet). Этот термин используется пока только внутри PowerShell. Командлет представляет собой класс .NET, порожденный от базового класса Cmdlet. Единый базовый класс Cmdlet гарантирует совместимый синтаксис всех командлетов, а также автоматизирует анализ параметров командной строки и описание синтаксиса командлетов, выдаваемое встроенной справкой.

Команды этого типа компилируются в динамическую библиотеку (DLL) и подгружаются к процессу PowerShell во время запуска оболочки (то есть сами по себе командлеты не могут быть запущены как приложения, но в них содержатся исполняемые объекты). Так как компилированный код подгружается к процессу оболочки, такие команды выполняются наиболее эффективно. Командлеты можно считать в определенном смысле аналогом внутренних команд традиционных оболочек, хотя в отличие от внутренних команд новые командлеты могут быть добавлены в систему в любое время.

> ***Замечание***
>
> Разрабатываются командлеты с помощью пакета PowerShell Software Developers Kit (SDK), который можно загрузить с официального сайта Microsoft. Подробное рассмотрение процесса создания командлетов выходит за рамки данной книги; мы в дальнейшем будем пользоваться только командлетами, входящими в стандартную поставку PowerShell.

Командлеты могут быть очень простыми или очень сложными, но каждый из них разрабатывается для решения одной узкой задачи. Работа с командлетами становится по-настоящему эффективной при использовании их композиции, когда объекты передаются от одного командлета другому по конвейеру (подробнее процедура конвейеризации обсуждается в *главе 5*).

Имена и структура командлетов

Имена командлетов всегда соответствуют шаблону "глагол-существительное", где глагол задает определенное действие, а существительное определяет объект, над которым это действие будет совершено. Это значительно упрощает запоминание и использование командлетов. Например, для получения ин-

формации о процессе служит командлет `Get-Process`, для остановки запущенной службы — командлет `Stop-Service`, для очистки экрана консоли — командлет `Clear-Host` и т. д.

Чтобы просмотреть список командлетов, доступных в ходе текущего сеанса, нужно выполнить командлет `Get-Command`:

```
PS C:\Documents and Settings\User> Get-Command

CommandType Name                    Definition
----------- ----                    ----------
Cmdlet      Add-Content             Add-Content [-Path] <Stri...
Cmdlet      Add-History             Add-History [[-InputObjec...
Cmdlet      Add-Member              Add-Member [-MemberType] ...
Cmdlet      Add-PSSnapin            Add-PSSnapin [-Name] <Str...
Cmdlet      Clear-Content           Clear-Content [-Path] <St...
Cmdlet      Clear-Item              Clear-Item [-Path] <Strin...
Cmdlet      Clear-ItemProperty      Clear-ItemProperty [-Path...
Cmdlet      Clear-Variable          Clear-Variable [-Name] <S...
Cmdlet      Compare-Object          Compare-Object [-Referenc...
Cmdlet      ConvertFrom-SecureString ConvertFrom-SecureString ...
Cmdlet      Convert-Path            Convert-Path [-Path] <Str...
Cmdlet      ConvertTo-Html          ConvertTo-Html [[-Propert...
Cmdlet      ConvertTo-SecureString  ConvertTo-SecureString [-...
Cmdlet      Copy-Item               Copy-Item [-Path] <String...
Cmdlet      Copy-ItemProperty       Copy-ItemProperty [-Path]...
Cmdlet      Export-Alias            Export-Alias [-Path] <Str...
Cmdlet      Export-Clixml           Export-Clixml [-Path] <St...
Cmdlet      Export-Console          Export-Console [[-Path] <...
Cmdlet      Export-Csv              Export-Csv [-Path] <Strin...
. . .
```

По умолчанию командлет `Get-Command` выводит сведения в трех столбцах: `CommandType` (тип команды), `Name` (имя) и `Definition` (определение). При этом в столбце `Definition` отображается синтаксис командлетов (многоточие (...) в столбце синтаксиса указывает на то, что данные обрезаны).

Командлеты могут иметь параметры — элементы, предоставляющие командлетам дополнительную информацию. Данная информация либо определяет элементы, с которыми должна работать команда, либо определяет, каким образом будет работать командлет. Параметры командлетов могут быть трех разных типов; обратиться к ним можно по имени, перед которым ставится

дефис (-), или по позиции (в последнем случае интерпретация параметра будет выполняться в зависимости от его местоположения в командной строке).

> **ЗАМЕЧАНИЕ**
>
> В командах оболочки cmd.exe для выделения имен параметров часто применялся символ / (например, `dir /s /b`); в оболочке PowerShell для параметров командлетов данный символ не используется.

В общем случае синтаксис командлетов имеет следующую структуру:

имя_командлета -параметр1 -параметр2 аргумент1 аргумент2

Здесь *-параметр1* — параметр, не имеющий значения (подобные параметры часто называют *переключателями*); *-параметр2* — имя параметра, имеющего значение *аргумент1*; *аргумент2* — параметр, не имеющий имени (или аргумент).

В качестве примера переключателя рассмотрим параметр `-Recurse` командлета `Get-ChildItem`.

> **ЗАМЕЧАНИЕ**
>
> Для краткости команд вместо командлета `Get-ChildItem` мы будем применять его псевдоним `dir` (более подробно вопросы, связанные с псевдонимами командлетов, обсуждаются в следующих разделах).

Переключатель `-Recurse`, если он указан, распространяет действие команды не только на определенный каталог, но и на все его подкаталоги. Например, следующий командлет выведет информацию обо всех файлах, которые находятся в каталоге c:\windows или его подкаталогах и имеют имя, удовлетворяющее маске n*d.exe (как видите, никакого аргумента после параметра `-Recurse` не указано):

```
PS C:\Documents and Settings\User> dir -Recurse -Filter n*d.exe c:\windows

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\windows

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          17.08.2004     16:04      69120 notepad.exe

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\windows\system32

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          17.08.2004     16:04      69120 notepad.exe
-a---          20.10.2001     16:00      31744 ntsd.exe
```

В данном примере используется еще один параметр `-Filter` с аргументом `n*d.exe`, задающим маску файлов для поиска. Отметим, что имена параметров не обязательно указывать полностью (однако сокращенное имя должно однозначно определять соответствующий параметр). Например, вместо последнего командлета можно выполнить следующий сокращенный вариант, результат останется тем же:

```
PS C:\Documents and Settings\User> dir -r -fi n*d.exe c:\windows
```

Однако если попытаться имя параметра `-Filter` сократить до одного символа, то возникнет ошибка:

```
PS C:\Documents and Settings\User> dir -r -f n*d.exe c:\windows
Get-ChildItem : Не удается обработать параметр, так как имя параметра "f"
неоднозначно. Возможные совпадения: -Filter -Force.
В строка:1 знак:4
+ dir  <<<< -recurse -f n*d.exe c:\windows
```

В нашем примере есть еще один аргумент, задающий путь к каталогу c:\windows. Как видите, параметр для этого аргумента не указан. На самом деле приведенная выше команда эквивалентна следующей:

```
PS C:\Documents and Settings\User>dir -r -f n*d.exe -Path c:\windows
```

То есть в данном случае аргументу `c:\windows` соответствует параметр `-Path`, однако командлет `Get-ChildItem` (псевдоним `dir`) устроен таким образом, что имя параметра `-Path` можно опускать — система сможет определить, что аргумент, задающий путь к каталогу файловой системы, относится именно к этому параметру.

> ***Замечание***
>
> Как видно из рассмотренных примеров, имена параметров не чувствительны к регистру. Мы будем писать сокращенные имена параметров со строчных букв, а имена, написанные полностью — с прописных.

Общие параметры командлетов

Напомним, что все командлеты являются потомками базового класса `Cmdlet`, в котором определены несколько параметров, обеспечивающих определенный уровень совместимости командлетов и согласования интерфейса оболочки PowerShell. Таким образом, некоторые параметры, называемые *общими*, поддерживаются всеми командлетами PowerShell (при этом на некоторые командлеты подобные параметры могут никак не влиять). Общие параметры перечислены в табл. 2.1.

Таблица 2.1. *Общие параметры командлетов PowerShell*

Параметр	Тип	Описание
-Verbose	Boolean	Включает режим вывода подробных сведений об операции, таких как результаты ее мониторинга или журналирование транзакций. Этот параметр эффективен только в командлетах, создающих подробные данные
-Debug	Boolean	Включает режим создания подробного отчета об операции на уровне программирования. Этот параметр эффективен только в командлетах, создающих данные отладки
-ErrorAction	Enum	Определяет, какой будет реакция командлета на возникновение ошибки (подробнее обработка ошибок обсуждается в *главе 9*). Возможные значения: Continue (по умолчанию), Stop, SilentlyContinue, Inquire
-ErrorVariable	String	Определяет переменную, в которой будут сохраняться ошибки команды во время выполнения. Эта переменная создается дополнительно к переменной $error
-OutVariable	String	Задает переменную, в которой будут сохраняться выходные данные команды во время выполнения
-OutBuffer	Int32	Определяет количество хранящихся в буфере объектов перед вызовом следующего командлета в конвейере
-WhatIf	Boolean	Предоставляет сведения об изменениях, которые произойдут в результате указанных действий, не производя самих этих действий. Данный параметр поддерживается командлетами в том случае, если они изменяют состояние системы
-Confirm	Boolean	Запрашивает разрешение у пользователя на выполнение каких-либо действий, вносящих изменения в систему. Данный параметр поддерживается командлетами в том случае, если они изменяют состояние системы

Скажем дополнительно несколько слов о параметре -WhatIf из табл. 2.1. Этот параметр позволяет увидеть объекты, на которые будет действовать тот или иной командлет, не выполняя при этом самих действий. Например, следующая команда позволяет увидеть, какие файлы в каталоге c:\temp будут удалены при выполнении команды del (сами файлы при этом не удаляются):

```
PS C:\> del -WhatIf "c:\temp\*.*"
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\bidiSNMP.dll".
```

```
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\BiDiSNMP.ini".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\Setup.exe".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XBASE.DLL".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XeroxLpr.dll".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XeroxPM.hlp".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XPmPrint.ini".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\xrxipdis.dll".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XSNMX.DLL".
What If: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\xv2p.dll".
```

Особенно полезным параметр `-WhatIf` может оказаться в тех случаях, когда диапазон объектов, на которые должен действовать командлет, определяется неявным образом (например, с помощью регулярных выражений).

Функции

Следующий тип команд PowerShell — функции. *Функция* — это блок кода на языке PowerShell, имеющий название и находящийся в памяти до завершения текущего сеанса командной оболочки. Анализ синтаксиса функции производится только один раз при ее объявлении (при повторном запуске функции подобный анализ не проводится).

Создадим простейшую функцию:

```
PS C:\Documents and Settings\User> function MyFunc{"Всем привет!"}
```

Эта функция имеет имя `MyFunc`, в ее теле просто выводится строка "Всем привет!". Вызовем нашу функцию из командной строки:

```
PS C:\Documents and Settings\User> MyFunc
Всем привет!
```

Функции, как и командлеты, поддерживают работу с параметрами (аргументами). Например, определим функцию `MyFunc1` с одним формальным параметром:

```
PS C:\Documents and Settings\User> function MyFunc1($a) {"Привет, $a!"}
```

Вызовем данную функцию с параметром:

```
PS C:\Documents and Settings\User> MyFunc1 Андрей
Привет, Андрей!
```

Более подробно вопросы, связанные с функциями PowerShell, рассмотрены в *главе 8*.

Сценарии

Третий тип команд, поддерживаемый PowerShell, — это сценарии. *Сценарий* представляет собой блок кода на языке PowerShell, хранящийся во внешнем файле с расширением ps1. Анализ синтаксиса сценария производится при каждом его запуске.

Сценарии позволяют работать с PowerShell в пакетном режиме, то есть заранее создать файл с нужными командами, определить логику работы с помощью различных управляющих инструкций языка PowerShell и пользоваться этим файлом как исполняемым модулем.

Более подробно вопросы, связанные со сценариями PowerShell, рассмотрены в *главе 8*.

Внешние исполняемые файлы

Последний тип команд, запускаемых в PowerShell, — внешние исполняемые файлы, которые выполняются операционной системой обычным образом. В частности, из оболочки PowerShell можно запускать любые внешние команды интерпретатора cmd.exe (например, `xcopy`), просто указывая их имена.

> ***Замечание***
>
> При необходимости можно использовать и внутренние команды cmd.exe; процедура вызова таких команд из оболочки PowerShell описана в *главе 18*.

Псевдонимы команд

Как отмечалось ранее, имена всех командлетов в PowerShell соответствуют шаблону "глагол-существительное" и могут быть довольно длинными, что затрудняет их быстрый набор. Механизм *псевдонимов*, реализованный в PowerShell, дает возможность пользователям выполнять команды по их альтернативным именам. В PowerShell заранее определено много псевдонимов, кроме того, можно добавлять в систему собственные псевдонимы.

Например, мы уже несколько раз пользовались командой `dir`, которая в действительности является псевдонимом командлета `Get-Childitem`. Убедиться в этом можно с помощью командлета `Get-Command` или `Get-Alias`:

```
PS C:\Documents and Settings\User> Get-Command dir

CommandType     Name                                Definition
-----------     ----                                ----------
Alias           dir                                 Get-ChildItem
PS C:\Documents and Settings\User> Get-Alias dir

CommandType     Name                                Definition
-----------     ----                                ----------
Alias           dir                                 Get-ChildItem
```

Псевдонимы в PowerShell делятся на два типа. Первый тип предназначен для совместимости имен с разными интерфейсами. Псевдонимы этого типа позволяют пользователям, имеющим опыт работы с другими оболочками (cmd.exe или UNIX-оболочки), использовать знакомые им имена команд для выполнения аналогичных операций в PowerShell. Это упрощает освоение новой оболочки, позволяя не тратить усилий на запоминание новых команд PowerShell. Например, пользователь хочет очистить экран. Если у него есть опыт работы с cmd.exe, то он, естественно, попробует выполнить команду `cls`. PowerShell при этом автоматически выполнит командлет `Clear-Host`, для которого `cls` является псевдонимом и который выполняет требуемое действие — очистку экрана. Для пользователей cmd.exe в PowerShell определены псевдонимы `cd`, `cls`, `copy`, `del`, `dir`, `echo`, `erase`, `move`, `popd`, `pushd`, `ren`, `rmdir`, `sort`, `type`; для пользователей UNIX — псевдонимы `cat`, `chdir`, `clear`, `diff`, `h`, `history`, `kill`, `lp`, `ls`, `mount`, `ps`, `pwd`, `r`, `rm`, `sleep`, `tee`, `write`.

Псевдонимы второго типа (стандартные псевдонимы) в PowerShell предназначены для быстрого ввода команд. Такие псевдонимы образуются из имен командлетов, которым они соответствуют. Например, глагол `Get` сокращается до `g`, глагол `Set` сокращается до `s`, существительное `Location` сокращается до `l` и т. д. Таким образом, командлету `Set-Location` соответствует псевдоним `sl`, а командлету `Get-Location` — псевдоним `gl`.

Просмотреть список всех псевдонимов, объявленных в системе, можно с помощью командлета `Get-Alias` без параметров:

```
PS C:\Documents and Settings\User> Get-Alias

CommandType     Name                                Definition
-----------     ----                                ----------
Alias           ac                                  Add-Content
```

```
Alias           asnp                Add-PSSnapin
Alias           clc                 Clear-Content
Alias           cli                 Clear-Item
Alias           clp                 Clear-ItemProperty
Alias           clv                 Clear-Variable
Alias           cpi                 Copy-Item
Alias           cpp                 Copy-ItemProperty
Alias           cvpa                Convert-Path
Alias           diff                Compare-Object
Alias           epal                Export-Alias
Alias           epcsv               Export-Csv
Alias           fc                  Format-Custom
Alias           fl                  Format-List
Alias           foreach             ForEach-Object
Alias           %                   ForEach-Object
Alias           ft                  Format-Table
Alias           fw                  Format-Wide
Alias           gal                 Get-Alias
Alias           gc                  Get-Content
Alias           gci                 Get-ChildItem
Alias           gcm                 Get-Command
Alias           gdr                 Get-PSDrive
Alias           ghy                 Get-History
Alias           gi                  Get-Item
Alias           gl                  Get-Location
Alias           gm                  Get-Member
Alias           gp                  Get-ItemProperty
Alias           gps                 Get-Process
Alias           group               Group-Object
. . .
```

Определить собственный псевдоним можно с помощью командлета `Set-Alias` или `New-Alias`. Например, создадим для командлета `Get-ChildItem` новый псевдоним `list`:

```
PS C:\Documents and Settings\User> Set-Alias -Name list -Value Get-
ChildItem
```

Проверим, как этот псевдоним работает:

```
PS C:\Documents and Settings\User> list

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\Documents
and Settings\User
```

```
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d---s          05.02.2008     7:54            Cookies
d----          12.05.2005    17:16            DoctorWeb
d----          06.11.2004    13:03            Phone Browser
d---s          22.09.2004    23:49            UserData
d-r--          04.10.2004    15:33            Главное меню
d-r--          05.12.2007     0:49            Избранное
d-r--          31.10.2007    21:03            Мои документы
d----          23.01.2008    11:13            Рабочий стол
-a---          24.05.2006    21:22       8304 gsview32.ini
```

Одной команде может соответствовать несколько псевдонимов (например, `Get-ChildItem` соответствуют псевдонимы `dir` и `ls`), но один псевдоним не может относиться к двум командам. Однако созданные псевдонимы можно переназначать, например:

```
PS C:\Documents and Settings\User> Set-Alias list Get-Location
PS C:\Documents and Settings\User> list
Path
----
C:\Documents and Settings\User
```

В данном примере мы назначили командлету `Get-Location` псевдоним `list`, который до этого был привязан к командлету `Get-ChildItem`. Также из последнего примера видно, что названия параметров `-Name` и `-Value` в командлете `Set-Alias` можно опустить, однако в этом случае следует соблюдать порядок указания имен псевдонима (первый параметр) и соответствующего ему командлета (второй параметр).

Псевдонимы в PowerShell можно создавать не только для командлетов, но и для функций, сценариев или исполняемых файлов. Например, команда `Set-Alias np c:\windows\notepad.exe` создаст псевдоним `np`, соответствующий исполняемому файлу notepad.exe (Блокнот Windows).

В PowerShell 1.0 нельзя создать псевдоним для команды с параметрами и значениями. Например, можно создать псевдоним для командлета `Set-Location`, но для команды `Set-Location c:\windows\system32` этого сделать нельзя. Чтобы назначить псевдоним подобной команде, можно создать функцию, включающую эту команду, и определить псевдоним для этой функции. Например:

```
PS C:\Documents and Settings\User> Function CD32 {Set-Location
c:\windows\system32}
PS C:\Documents and Settings\User> Set-Alias go cd32
```

```
PS C:\Documents and Settings\User> go
PS C:\windows\system32>
```

Удалить псевдоним можно с помощью командлета `Remove-Item`, например:

```
PS C:\windows\system32> Remove-Item alias:go
PS C:\windows\system32> go
Условие "go" не распознано как командлет, функция, выполняемая программа
или файл сценария. Проверьте условие и повторите попытку.
В строка:1 знак:2
+ go <<<<
```

Псевдонимы можно экспортировать в текстовый файл (командлет `Export-Alias`) и импортировать их из файла (командлет `Import-Alias`).

Благодаря псевдонимам синтаксис PowerShell приобретает гибкость: одни и те же команды могут быть записаны и очень кратко, и в развернутом виде. Это очень важно для языка, который одновременно является оболочкой командной строки и языком написания сценариев. При интерактивной работе для ускорения ввода команд удобнее применять краткие псевдонимы и не полностью указывать имена параметров. Если же вы пишете сценарии, то лучше использовать полные названия командлетов и их параметров, это значительно упростит в дальнейшем разбор программного кода.

Диски PowerShell

Все мы давно привыкли к структуре файловой системы как совокупности вложенных папок (каталогов) и файлов. В операционной системе Windows интерфейс к такой структуре предоставляют Проводник Windows, оболочка cmd.exe, а также различные файловые менеджеры сторонних разработчиков. В UNIX-системах понятия файлов и папок трактуются более широко, в качестве файлов здесь могут выступать различные компоненты системы (например, аппаратные устройства или сетевые подключения). Такой подход упрощает поиск нужных элементов в операционной системе. Оболочки командной строки или другие утилиты, обращающиеся к файлам в UNIX-системах, могут работать также и с этими компонентами.

Оболочка PowerShell в этом аспекте похожа на UNIX-системы, так как она позволяет просматривать различные хранилища данных и перемещаться по ним с использованием тех же привычных процедур, которые применяются для перемещения по файловой системе. Помимо обычных локальных или сетевых дисков файловой системы (`C:`, `D:` и т. д.) оболочка поддерживает специальные (виртуальные) *диски PowerShell*, связанные с хранилищами

данных разных типов. Например, корневому разделу реестра HKEY_LOCAL_MACHINE соответствует диск `HKLM:`, псевдонимам, доступным в текущем сеансе работы, соответствует диск `Alias:`, а хранилищу сертификатов цифровых подписей — диск `Cert:`.

> ***Замечание***
>
> В отличие от обычных локальных или сетевых дисков файловой системы диски PowerShell доступны только из оболочки PowerShell, обратиться к ним из Проводника Windows нельзя. Имена дисков PowerShell могут содержать более одного символа.

Для получения списка дисков PowerShell, доступных в текущем сеансе работы, нужно воспользоваться командлетом `Get-PSDrive`:

```
PS C:\> Get-PSDrive

Name           Provider      Root                 CurrentLocation
----           --------      ----                 ---------------
A              FileSystem    A:\
Alias          Alias
C              FileSystem    C:\
Cert           Certificate   \
D              FileSystem    D:\
Env            Environment
Function       Function
HKCU           Registry      HKEY_CURRENT_USER
HKLM           Registry      HKEY_LOCAL_MACHINE
Variable       Variable
```

Как видите, командлет `Get-PSDrive` для каждого диска сообщает имя провайдера (колонка `Provider`), поддерживающего этот диск.

Провайдеры PowerShell

Провайдер PowerShell — это .NET-приложение, предоставляющее пользователям PowerShell доступ к данным из определенного специализированного хранилища в согласованном формате, напоминающем формат обычных дисков файловой системы. Тем самым провайдеры PowerShell обеспечивают доступ к данным, к которым трудно обратиться через командную строку иными способами.

В оболочку PowerShell по умолчанию включено несколько встроенных провайдеров, которые можно использовать для доступа к различным хранилищам данных (табл. 2.2).

Таблица 2.2. Встроенные провайдеры PowerShell

Провайдер	Хранилище данных
Alias	Псевдонимы PowerShell
Certificate	Сертификаты X509 для цифровых подписей
Environment	Переменные среды Windows
FileSystem	Диски файловой системы, каталоги и файлы
Function	Функции PowerShell
Registry	Реестр Windows
Variable	Переменные PowerShell

Дополнительно к встроенным, можно создавать собственные провайдеры PowerShell и устанавливать провайдеры, созданные другими разработчиками (например, для доступа к каталогам Active Directory или к почтовым ящикам Microsoft Exchange).

> ***Замечание***
>
> Новые провайдеры добавляются в оболочку PowerShell путем установки специальных оснасток PowerShell, в которых также могут находиться дополнительные командлеты. Подробное рассмотрение вопросов установки дополнительных оснасток PowerShell выходит за рамки данной книги, дополнительную информацию можно получить из справочной системы PowerShell, выполнив команду `Get-Help about_pssnapins`.

Для просмотра списка зарегистрированных в оболочке PowerShell провайдеров нужно воспользоваться командлетом `Get-PSProvider`:

```
PS C:\> Get-PSProvider

Name                 Capabilities                Drives
----                 ------------                ------
Alias                ShouldProcess               {Alias}
Environment          ShouldProcess               {Env}
FileSystem           Filter, ShouldProcess       {C, D, E, F...}
Function             ShouldProcess               {Function}
Registry             ShouldProcess               {HKLM, HKCU}
Variable             ShouldProcess               {Variable}
Certificate          ShouldProcess               {cert}
```

Навигация по дискам PowerShell

Основное назначение провайдеров заключается в том, что они обеспечивают доступ к разнородным данным привычным согласованным образом. Используемая при этом модель представления данных основана на дисках файловой системы.

Предлагаемые провайдером данные можно просматривать и изменять так, как если бы они хранились в виде каталогов и файлов на жестком диске. Навигация по различным дискам PowerShell и просмотр содержимого этих дисков осуществляется с помощью одних и тех же базовых командлетов.

Работая с файловой системой, мы используем понятие *текущего* или *рабочего каталога*. К файлам в рабочем каталоге можно обращаться по имени, не указывая полного пути к ним. В оболочке cmd.exe для определения или смены рабочего каталога служит команда `cd` (можно пользоваться и полным именем `chdir`):

```
C:\Documents and Settings\404_Popov> cd /?
Вывод имени либо смена текущего каталога.
CHDIR [/D] [диск:][путь]
CHDIR [..]
CD [/D] [диск:][путь]
CD [..]
  ..  обозначает переход в родительский каталог.

Команда CD диск: отображает имя текущего каталога указанного диска.
Команда CD без параметров отображает имена текущих диска и каталога.
. . .
```

В оболочке PowerShell понятие рабочего (текущего) каталога распространяется и на диски PowerShell. Узнать путь к текущему каталогу можно с помощью командлета `Get-Location` (псевдоним `pwd` данного командлета соответствует команде UNIX-оболочек с аналогичной функциональностью):

```
PS C:\Documents and Settings\404_Popov> Get-Location

Path
----
C:\Documents and Settings\404_Popov
```

Для смены текущего каталога (в том числе для перехода на другой диск PowerShell) используется командлет `Set-Location` (псевдонимы `cd`, `chdir`, `sl`).

Например:

```
PS C:\Documents and Settings\404_Popov> Set-Location c:\
PS C:\> Set-Location HKLM:\Software
PS HKLM:\Software>
```

Как видите, при вводе командлета `Set-Location` на экран явно не выводится отзыв о его выполнении. При необходимости можно использовать параметр `PassThru`, выводящий после выполнения команды `Set-Location` путь к текущему каталогу:

```
PS HKLM:\Software> Set-Location 'C:\Program Files' -PassThru
Path
----
C:\Program Files
```

В оболочке cmd.exe, как и в оболочках UNIX-систем, поддерживаются абсолютные и относительные пути. Первые задают полный путь, тогда как вторые указываются относительно рабочего каталога. При этом текущему каталогу соответствует путь . (точка), родительскому каталогу текущего каталога — путь .. (две точки), а корневому каталогу текущего диска — путь \ (обратная косая черта). В PowerShell данная нотация сохраняется. Например (вместо командлета `Set-Location` мы используем его псевдоним `cd`):

```
PS C:\Program Files> cd \ -PassThru
Path
----
C:\

PS C:\> cd HKLM:\Software -PassThru
Path
----
HKLM:\Software

PS HKLM:\Software> cd .. -PassThru
Path
----
HKLM:\
```

Просмотр содержимого дисков и каталогов

Для просмотра элементов и контейнеров, находящихся на определенном диске PowerShell, можно воспользоваться командлетом `Get-ChildItem` (псевдонимы `dir` и `ls`). Естественно, выводимая информация при этом будет зависеть

от типа диска PowerShell. Для примера выполним командлет Get-ChildItem на диске, соответствующем корневому разделу реестра HKEY_CURRENT_USER, и на обычном диске файловой системы:

```
PS C:\Documents and Settings\404_Popov> cd hkcu:
PS HKCU:\> dir

   Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER

SKC  VC Name                           Property
---  -- ----                           --------
  2   0 AppEvents                      {}
  1  32 Console                        {ColorTable00, Color...}
 24   1 Control Panel                  {Opened}
  0   2 Environment                    {TEMP, TMP}
  1   5 Identities                     {Identity Ordinal, Mig...}
  4   0 Keyboard Layout                {}
 10   0 Network                        {}
  4   1 Printers                       {DeviceOld}
 46   0 Software                       {}
  0   0 UNICODE Program Groups         {}
  2   0 Windows 3.1 Migration Status   {}
  0   1 SessionInformation             {ProgramCount}
  0   7 Volatile Environment           {LOGONSERVER, CLIENTNA...}

PS HKCU:\> cd 'C:\Program Files'
PS C:\Program Files> ls

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\Program Files

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         18.06.2006     13:29            Adobe
d----         18.06.2006     15:45            Borland
d----         17.01.2007     17:43            Common Files
d----         18.05.2006     11:58            ComPlus Applications
. . .
```

Командлет Get-ChildItem имеет несколько параметров, позволяющих, в частности, просматривать содержимое вложенных каталогов, отображать скры-

тые элементы, применять фильтры для отображения файлов по маске и т. д. Более подробную информацию о возможностях и параметрах командлета `Get-ChildItem` можно найти в справочной системе PowerShell (см. *главу 3*).

Создание дисков

Помимо использования стандартных дисков PowerShell, с помощью командлета `New-PSDrive` можно создавать собственные пользовательские диски. Для этого в командлете `New-PSDrive` нужно указать три параметра: `-Name` (имя создаваемого диска PowerShell), `-PSProvider` (название провайдера, например `FileSystem` для диска файловой системы или `Registry` для диска, соответствующего раздела реестра) и путь к корневому каталогу нового диска.

Например, можно создать диск для определенной папки на жестком диске для того, чтобы обращаться к ней не по "настоящему" длинному пути, а просто по имени диска. Создадим диск PowerShell с именем `Docs`, который будет соответствовать папке с документами определенного пользователя:

```
PS C:\Program Files> New-PSDrive -Name Docs -PSProvider FileSystem -Root
'C:\Documents and Settings\404_Popov\Мои документы'

Name  Provider     Root                                CurrentLocation
----  --------     ----                                ---------------
Docs  FileSystem   C:\Documents and Settings\4...
```

Теперь обращаться к новому диску можно точно так же, как и к другим дискам PowerShell:

```
PS C:\Program Files> cd docs: -PassThru
Path
----
Docs:\


PS Docs:\> dir

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\Documents
and Settings\404_Popov\Мои документы

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         20.10.2006     15:11            111
d----         31.07.2006      8:06            CyberLink
d----         18.06.2006     18:04            My eBooks
d----         14.11.2006      9:13            My PaperPort Documents
d----         14.02.2008     23:35            WindowsPowerShell
```

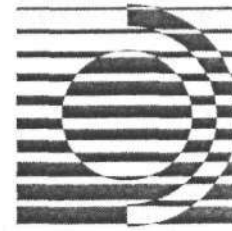
```
d-r--         19.11.2007     14:16            Мои рисунки
d-r--         14.06.2006     16:08            Моя музыка
. . .
```

В качестве еще одного примера создадим пользовательский диск `CurrVer` для ветви реестра HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion, где хранятся различные важные параметры операционной системы:

```
PS Docs:\> New-PSDrive -Name CurrVer -PSProvider Registry -Root
HKLM\Software\Microsoft\Windows\CurrentVersion

Name       Provider   Root                           CurrentLocation
----       --------   ----                           ---------------
CurrVer    Registry   HKLM\Software\Microsoft\...
```

Теперь содержимое ветви реестра можно просматривать, не вводя длинного пути, трудного для запоминания:

```
PS Docs:\> dir CurrVer:\

   Hive: Microsoft.PowerShell.Core\Registry::HKLM\Software\Microsoft
\Windows\CurrentVersion

SKC  VC Name                           Property
---  -- ----                           --------
  4   0 App Management                 {}
 56   0 App Paths                      {}
  1   0 Applets                        {}
  0   0 BITS                           {}
  5   0 Control Panel                  {}
  5   0 Controls Folder                {}
  0   0 CSCSettings                    {}
  1   0 DateTime                       {}
  1   0 Dynamic Directory              {}
 35   1 Explorer                       {IconUnderline}
  0   3 Extensions                     {.ini, .txt, .wtx}
. . .
```



Глава 3

Приемы работы в оболочке

При работе в командной оболочке нам приходится вручную вводить команды, которые могут быть довольно длинными и иметь много разных параметров, которые трудно запомнить. Поэтому важно научиться грамотно пользоваться справочной системой, имеющейся в системе, а также использовать возможности оболочки для быстрого набора команд или повторного вызова уже вводившихся команд.

Редактирование в командном окне PowerShell

В PowerShell поддерживаются те же возможности редактирования вводимых команд, что и в оболочке cmd.exe (табл. 3.1).

Таблица 3.1. *Возможности редактирования в командной строке PowerShell*

Клавиатурная комбинация	Действие
<←>	Перемещение курсора на один символ влево
<→>	Перемещение курсора на один символ вправо
<Ctrl>+<←>	Перемещение курсора влево на одно слово
<Ctrl>+<→>	Перемещение курсора вправо на одно слово
<Home>	Перемещение курсора в начало текущей строки
<End>	Перемещение курсора в конец текущей строки
<↑>/<↓>	Просмотр истории команд
<Insert>	Переключение между режимами вставки и замены

Таблица 3.1 (окончание)

Клавиатурная комбинация	Действие
<Delete>	Удаление символа, на который указывает курсор
<Backspace>	Удаление символа, находящегося перед курсором
<F7>	Вывод на экран окна со списком вводившихся в текущем сеансе команд. С помощью клавиш управления курсором можно выбрать нужную команду, нажать <Enter>, и данная команда будет выполнена
<Tab>	Автоматическое завершение команды (подробнее см. *раздел "Автоматическое завершение команд"*)

При работе с оболочкой PowerShell можно пользоваться буфером Windows. Выделить и скопировать текст в буфер из окна PowerShell можно следующим образом: щелкнуть правой кнопкой мыши на заголовке окна PowerShell (или любой кнопкой мыши на пиктограмме окна), последовательно выбрать пункты **Изменить** и **Пометить** в появившемся контекстном меню, затем с помощью клавиш управления курсором, удерживая нажатой клавишу <Shift>, выделить нужный блок текста и нажать клавишу <Enter>. Все содержимое окна PowerShell можно выделить, выбрав пункты **Изменить** и **Выделить все** того же контекстного меню. Вставка текста из буфера Windows в командное окно PowerShell (в текущую позицию курсора) осуществляется с помощью выбора пунктов **Изменить** и **Вставить**.

Можно упростить манипуляции с буфером Windows, включив режимы выделения мышью и быстрой вставки. Для этого нужно щелкнуть правой кнопкой мыши на заголовке окна PowerShell, выбрать в контекстном меню пункт **Свойства** и установить флажки **Выделение мышью** и **Быстрая вставка** в секции **Редактирование** на закладке **Общие** диалогового окна **Свойства Windows PowerShell** (рис. 3.1).

После этого выделять текст можно мышью, удерживая нажатой левую кнопку. Для копирования выделенного фрагмента в буфер достаточно нажать клавишу <Enter> или щелкнуть правой кнопкой мыши. Вставка текста из буфера в текущую позицию курсора производится также путем нажатия правой кнопки мыши.

> ***Замечание***
>
> Если у вас есть сценарий PowerShell (внешний текстовый файл с командами PowerShell), содержащий несколько строк, то можно копировать в буфер Windows и вставлять в окно PowerShell сразу все команды (они будут выполняться по очереди), а не по одной.

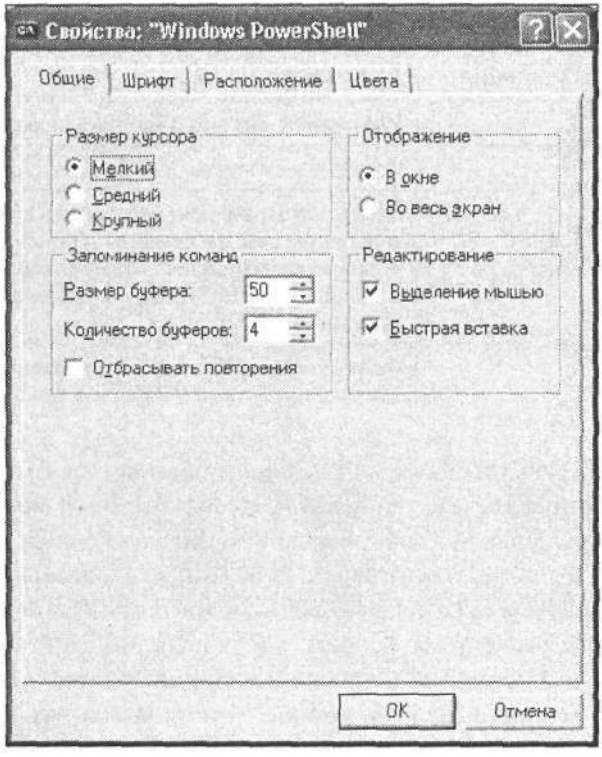


Рис. 3.1. Настройка свойств окна PowerShell

На самом деле, большинство из приведенных в табл. 3.1 клавиатурных комбинаций действуют одинаково во всех консольных приложениях Windows; манипуляции с буфером Windows также выполняются одинаково во всех текстовых окнах. Однако в PowerShell реализована одна новая важная возможность редактирования — это автоматическое завершение команд и их параметров при вводе.

Автоматическое завершение команд

Находясь в оболочке PowerShell, можно ввести часть какой-либо команды, нажать клавишу <Tab>, и система попытается сама завершить эту команду.

Во-первых, автоматическое завершение срабатывает для имен файлов и путей файловой системы (подобный режим поддерживался и в оболочке cmd.exe). При нажатии клавиши <Tab> PowerShell автоматически расширит частично введенный путь файловой системы до первого найденного совпадения. При повторном нажатии клавиши <Tab> производится циклический

переход по имеющимся возможностям выбора. Например, нам нужно переместиться в каталог 'C:\Program Files'. В оболочке cmd.exe навигация по файловой системе осуществлялась при помощи команды `cd`. Введем данную команду в PowerShell и в качестве параметра укажем начало имени нужного нам каталога:

```
PS C:\Documents and Settings\User> cd c:\pro
```

Нажмем теперь клавишу <Tab>, и система автоматически дополнит путь к каталогу:

```
PS C:\Documents and Settings\User> cd 'C:\Program Files'
```

Как видите, название каталога, содержащее пробелы, заключается в апострофы.

Также в PowerShell реализована возможность автоматического завершения путей файловой системы на основе шаблонных символов: ? (заменяет один произвольный символ) и * (заменяет любое количество произвольных символов). Например, если в PowerShell ввести команду `cd c:\pro*files` и нажать клавишу <Tab>, то в строке ввода вновь появится команда `cd 'C:\Program Files'`.

Во-вторых, в PowerShell реализовано автоматическое завершение имен командлетов и их параметров. Если ввести первую часть имени командлета (глагол) и дефис, нажать после этого клавишу <Tab>, то система подставит имя первого подходящего командлета (следующий вариант имени выбирается путем повторного нажатия <Tab>). Например, введем в командной строке PowerShell глагол `get-`:

```
PS C:\Documents and Settings\User> get-
```

Нажмем клавишу <Tab>:

```
PS C:\Documents and Settings\User> Get-Acl
```

Еще раз нажмем клавишу <Tab>:

```
PS C:\Documents and Settings\User> Get-Alias
```

Нажимая далее клавишу <Tab>, мы можем перебрать все командлеты, начинающиеся с глагола `Get`.

Аналогичным образом автоматическое завершение срабатывает для частично введенных имен параметров командлета: нажимая клавишу <Tab>, мы будем циклически перебирать подходящие имена. Например, введем в командной строке PowerShell имя командлета, за которым через пробел следует знак дефиса:

```
PS C:\Documents and Settings\User> Get-Alias -
```

Нажмем клавишу <Tab>:

```
PS C:\Documents and Settings\User> Get-Alias -Name
```

Как видите, система подставила в нашу команду параметр -Name. Еще раз нажмем клавишу <Tab>:

```
PS C:\Documents and Settings\User> Get-Alias -Exclude
```

Нажимая далее клавишу <Tab>, мы можем перебрать все параметры, поддерживающиеся командлетом Get-Alias.

В-третьих, PowerShell позволяет автоматически завершать имена используемых переменных. Например, создадим переменную $StringVariable:

```
PS C:\Documents and Settings\User> $StringVariable='asdfg'
```

Введем теперь часть имени этой переменной:

```
PS C:\Documents and Settings\User> $str
```

Нажав клавишу <Tab>, мы получим в командной строке полное имя нашей переменной:

```
PS C:\Documents and Settings\User> $StringVariable
```

Наконец, PowerShell поддерживает автоматическое завершение имен свойств и методов объектов (аналогичные возможности имелись ранее только в специальных системах для разработки приложений типа Microsoft Visual Studio или Borland Delphi). Например, введем следующие команды:

```
PS C:\Documents and Settings\User> $s='qwerty'
PS C:\Documents and Settings\User> $s.len
```

Нажмем клавишу <Tab>:

```
PS C:\Documents and Settings\User> $s.Length
```

Как видите, система подставила свойство Length, имеющееся у символьных переменных. Если подставляется имя метода (функции), а не свойства, то после его имени автоматически ставится круглая скобка. Например, введем следующую команду:

```
PS C:\Documents and Settings\User> $s.sub
```

Нажмем клавишу <Tab>:

```
PS C:\Documents and Settings\User> $s.Substring(
```

Теперь можно вводить параметры метода Substring.

Справочная система PowerShell

При работе с интерактивной командной оболочкой очень важно иметь под рукой подробную и удобную справочную систему с описанием возможностей команд и примерами их применения. В PowerShell такая система имеется, здесь предусмотрено несколько способов получения справочной информации внутри оболочки.

Получение справки о командлетах

Краткую справку по какому-либо одному командлету можно получить с помощью параметра -? (вопросительный знак), указанного после имени этого командлета. Например, для получения справки по командлету Get-Process нужно выполнить следующую команду:

```
PS C:\> Get-Process -?

ИМЯ
    Get-Process

ОПИСАНИЕ
    Отображает процессы, выполняющиеся на локальном компьютере.

СИНТАКСИС
    Get-Process [[-name] <string[]>] [<CommonParameters>]
    Get-Process -id <Int32[]> [<CommonParameters>]
    Get-Process -inputObject <Process[]> [<CommonParameters>]

ПОДРОБНОЕ ОПИСАНИЕ
    Командлет Get-Process извлекает объект-процесс для каждого процесса.
При использовании командлета "Get-Process" без указания параметров
происходит отображение всех процессов, выполняющихся на компьютере, что
эквивалентно команде "Get-Process *". Процесс можно определить по имени
или идентификатору (PID) или передав объект по конвейеру в командлет
Get-Process. Для Get-Process по умолчанию передается имя процесса.
Для Stop-Process по умолчанию передается идентификатор процесса.

ССЫЛКИ ПО ТЕМЕ
    Stop-Process

ЗАМЕЧАНИЯ
    Для получения дополнительных сведений введите: "get-help Get-Process
-detailed".
    Для получения технических сведений введите: "get-help Get-Process
-full".
```

Как видите, в этой справке кратко описывается интересующий нас командлет и приводятся допустимые варианты его синтаксиса. Необязательные параметры выводятся в квадратных скобках. Если для параметра необходимо указывать аргумент определенного типа, то после имени такого параметра в угловых скобках приводится название этого типа.

Для получения подробной информации о командлете служит специальный командлет Get-Help, который следует запускать с параметрами -Detailed или -Full. Ключ -Full приводит к выводу всей имеющейся справочной информации, а при использовании ключа -Detailed некоторая техническая информация опускается. В обоих случаях будут выведены подробные описания каждого из параметров, поддерживаемых рассматриваемым командлетом,

различные замечания, а также приведены примеры запуска данного командлета с различными параметрами и аргументами. Например:

```
PS C:\> Get-Help Get-Process -Full

ИМЯ
    Get-Process
ОПИСАНИЕ
    Отображает процессы, выполняющиеся на локальном компьютере.
СИНТАКСИС
    Get-Process [[-name] <string[]>] [<CommonParameters>]
    Get-Process -id <Int32[]> [<CommonParameters>]
    Get-Process -inputObject <Process[]> [<CommonParameters>]
ПОДРОБНОЕ ОПИСАНИЕ
    Командлет Get-Process извлекает объект-процесс для каждого
    процесса. При использовании командлета "Get-Process" без
    указания параметров происходит отображение всех процессов,
    выполняющихся на компьютере, что эквивалентно команде
    "Get-Process *". Процесс можно определить по имени или
    идентификатору (PID) или передав объект по конвейеру в командлет
    Get-Process. Для Get-Process по умолчанию передается имя
    процесса. Для Stop-Process по умолчанию передается идентификатор
    процесса.
ПАРАМЕТРЫ
    -name <string[]>
        Задает один или несколько процессов, используя их имена.
        Можно ввести несколько имен процессов, разделяя их запятыми,
        либо использовать подстановочные знаки. Параметр ("-Name"),
        задающий имя, является необязательным.

        Требуется?                    false
        Позиция?                      1
        Значение по умолчанию         Null
        Принимать входные данные конвейера? true (ByPropertyName)
        Принимать подстановочные знаки?  true
. . .
```

Как видите, в описании параметра `Name` даются сведения о пяти атрибутах (табл. 3.2). Эти атрибуты характерны для большинства параметров командлетов.

Таблица 3.2. Атрибуты параметров командлетов PowerShell

Параметр	Описание
Требуется?	Указывает, будет ли командлет выполняться при отсутствии этого параметра. Если настройке присвоено значение `True`, то при запуске данного командлета необходимо указывать параметр. Если параметр не задан, система запросит его значение
Позиция?	Определяет, можно ли задавать значение параметра без указания его имени, и позицию, в которой он должен быть указан, если это возможно. Если атрибут имеет значение `0` или `named`, то при задании значения параметра необходимо указывать его имя (напомним, что параметр данного типа называется именованным). Именованные параметры могут перечисляться после имени командлета в любом порядке. Если атрибут "Позиция?" имеет целое ненулевое значение, то имя параметра указывать не обязательно (позиционный параметр). Значение атрибута "Позиция?" определяет порядковый номер параметра в списке других позиционных параметров. При указании имени позиционные параметры могут перечисляться после имени командлета в любом порядке
Значение по умолчанию?	Содержит значение по умолчанию, которое используется в том случае, когда никакого иного значения не указано. Обязательным параметрам, так же как и многим необязательным, никогда не присваивается значение по умолчанию. Например, многие команды, чьим входным значением является параметр `-Path`, при отсутствии соответствующего значения используют текущее местоположение
Принимать входные данные конвейера?	Определяет, может ли параметр получать свое значение из объекта в конвейере. Чтобы команду можно было включить в конвейер, соответствующая настройка ее входного параметра должна иметь значение `True`, что дает возможность принимать конвейерный ввод
Принимать подстановочные знаки?	Показывает, может ли значение параметра содержать подстановочные знаки, что дает возможность сопоставлять его с несколькими существующими в целевом контейнере элементами

Справочная информация, не связанная с командлетами

Все доступные разделы справочной системы PowerShell можно увидеть с помощью команды `Get-Help *`:

```
PS C:\Documents and Settings\User> Get-Help *

Name                              Category  Synopsis
----                              --------  --------
ac                                 Alias    Add-Content
asnp                               Alias    Add-PSSnapin
```

```
clc                             Alias     Clear-Content
cli                             Alias     Clear-Item
clp                             Alias     Clear-ItemProperty
clv                             Alias     Clear-Variable
. . .
set                             Alias     Set-Variable
type                            Alias     Get-Content
Get-Command                     Cmdlet    Возвращает базовые сведен...
Get-Help                        Cmdlet    Отображает сведения о ком...
Get-History                     Cmdlet    Возвращает список команд,...
. . .
Trace-Command                   Cmdlet    Командлет Trace-Command н...
Alias                           Provider  Предоставляет доступ к пс...
Environment                     Provider  Предоставляет доступ к пе...
FileSystem                      Provider  Поставщик PowerShell для ...
. . .
Certificate                     Provider  Обеспечивает доступ к хра...
about_alias                     HelpFile  Использование альтернатив...
about_arithmetic_operators      HelpFile  Операторы, которые исполь...
about_array                     HelpFile  Компактная структура разм...
. . .
```

Как видите, командлет `Get-Help` позволяет просматривать справочную информацию не только о разных командлетах, но и о синтаксисе языка PowerShell, о псевдонимах, провайдерах, функциях и других аспектах работы оболочки. Список тем, обсуждение которых представлено в справочной службе PowerShell, можно увидеть с помощью следующей команды:

```
PS C:\Documents and Settings\User> Get-Help about_*

Name                            Category Synopsis
----                            -------- --------
about_alias                     HelpFile Использование альтернативн...
about_arithmetic_operators      HelpFile Операторы, которые использ...
about_array                     HelpFile Компактная структура разме...
about_assignment_operators      HelpFile Операторы, используемые в ...
about_associative_array         HelpFile Компактная структура данны...
about_automatic_variables       HelpFile Переменные, автоматически ...
about_break                     HelpFile Инструкция для немедленног...
about_command_search            HelpFile Как оболочка Windows Power...
```

```
about_command_syntax          HelpFile Формат команд в Windows Po...
about_commonparameters        HelpFile Параметры, поддерживаемые ...
. . .
```

Таким образом, чтобы прочитать информацию об использовании массивов в PowerShell, нужно выполнить следующую команду:

```
PS C:\Documents and Settings\User> Get-Help about_array
РАЗДЕЛ
    Массивы
КРАТКОЕ ОПИСАНИЕ
  Компактная структура размещения элементов данных
ПОЛНОЕ ОПИСАНИЕ
  Массив представляет собой структуру данных для размещения набора
  элементов данных одного типа. Оболочка Windows PowerShell
  поддерживает такие элементы данных, как string, int
  (32-разрядное целое число), long (64-разрядное целое число),
  bool (логическое значение), byte, а также другие типы объектов
  .NET.
СОЗДАНИЕ И ИНИЦИАЛИЗАЦИЯ МАССИВА
  Чтобы создать и инициализировать массив, присвойте несколько
  значений переменной. Значения, введенные в массив, разделяются
  запятой и отделены от имени переменной оператором присваивания(=).
. . .
```

Для просмотра списка статей справки по определенной тематике нужно воспользоваться параметром `-Category` командлета `Get-Help`. Например, если требуется найти информацию о провайдерах PowerShell, то нужно выполнить следующую команду:

```
PS C:\Documents and Settings\User> Get-Help -Category provider

Name          Category   Synopsis
----          --------   --------
Alias         Provider   Предоставляет доступ к псевдонимам W...
Environment   Provider   Предоставляет доступ к переменным ср...
FileSystem    Provider   Поставщик PowerShell для доступа к ф...
Function      Provider   Предоставляет доступ к функциям, опр...
Registry      Provider   Предоставляет доступ к разделам и зн...
Variable      Provider   Предоставляет доступ к переменным Wi...
Certificate   Provider   Обеспечивает доступ к хранилищам сер...
```

Чтобы получить сведения о конкретном провайдере (например, о провайдере реестра Windows), введите следующую команду:

```
PS C:\Documents and Settings\User> Get-Help registry

ИМЯ ПОСТАВЩИКА
    Registry

ДИСКИ
    HKLM, HKCU

ОПИСАНИЕ
    Предоставляет доступ к разделам и значениям системного реестра
    в Windows PowerShell.

ПОДРОБНОЕ ОПИСАНИЕ
    Поставщик Windows PowerShell Registry позволяет извлекать,
    добавлять, изменять, очищать и удалять разделы и
    значения реестра в Windows PowerShell.

    Разделы реестра представлены как экземпляры класса
    Microsoft.Win32.RegistryKey. Значения реестра представлены как
    экземпляры класса PSCustomObject.
. . .
```

Отметим, что командлет `Get-Help` выводит содержимое раздела справки на экран сразу целиком. Функции `man` и `help` позволяют выводить справочную информацию поэкранно (аналогично команде `more` интерпретатора `cmd.exe`), например: `man about_array`.

История команд в сеансе работы

Информацию обо всех командах, которые мы выполняем в оболочке PowerShell, система записывает в специальный *журнал сеанса* или *журнал команд*, что дает возможность повторно использовать эти команды, не набирая их полностью на клавиатуре. Журнал сеанса сохраняется до выхода из оболочки PowerShell.

По умолчанию PowerShell сохраняет 64 последние команды. Для изменения числа сохраняемых команд нужно изменить значение специальной переменной `$MaximumHistoryCount`, например:

```
PS C:\> $MaximumHistoryCount
64
```

```
PS C:\> $MaximumHistoryCount=100
PS C:\> $MaximumHistoryCount
100
```

Для передвижения назад по журналу команд следует нажимать клавишу <↑>. При первом нажатии на эту клавишу в командной строке будет отображена последняя команда, выполнявшаяся в текущем сеансе работы. При повторном нажатии появится предпоследняя выполненная команда и так далее. Найдя нужную команду, можно отредактировать ее, после чего нажать <Enter>, чтобы запустить на выполнение. Клавиша <↓> позволяет перемещаться по журналу команд вперед.

Можно просматривать не все команды в журнале сеанса, а только команды, начинающиеся с определенных символов. Для этого нужно ввести эти начальные символы в командной строке и нажимать клавишу <F8>.

Полностью содержимое журнала команд можно отобразить на экране, нажав клавишу <F7> (рис. 3.2).

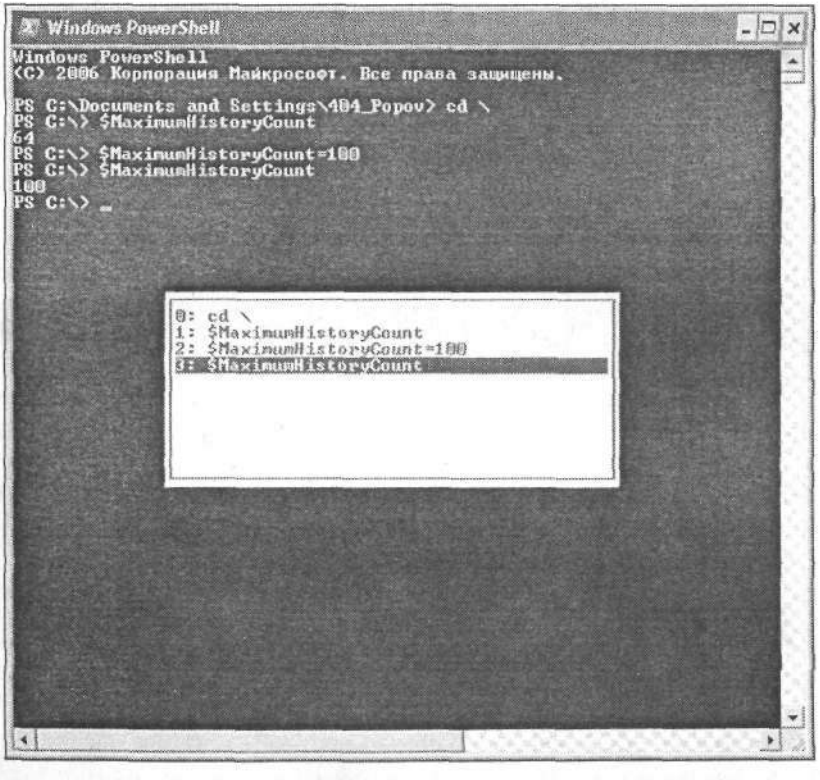


Рис. 3.2. Журнал команд PowerShell

В данном прокручиваемом списке можно с помощью клавиш управления курсором выбрать нужную команду, нажать <Enter> и данная команда будет выполнена.

Также можно выполнить команду по ее порядковому номеру. Для этого следует нажать клавишу <F9>, ввести нужный номер и нажать <Enter> (рис. 3.3).

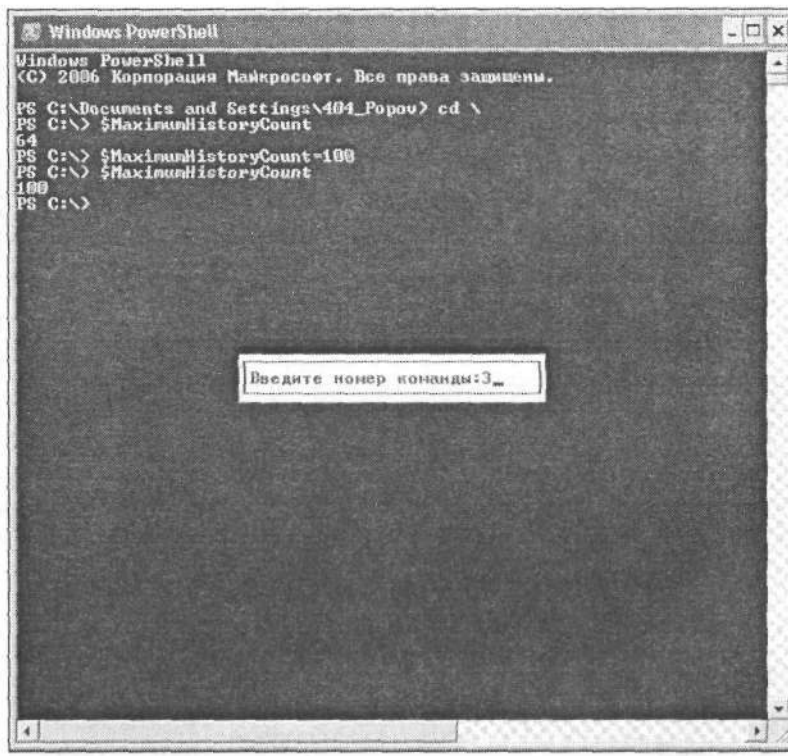


Рис. 3.3. Вызов команды по ее номеру

В дополнение к клавиатурным комбинациям в PowerShell имеются специальные командлеты для работы с журналом команд. Команда `Get-History` (псевдонимы `h`, `history` и `ghy`) позволяет вывести историю команд:

```
PS C:\> Get-History

  Id CommandLine
  -- -----------
   1 cd C:
   2 cd \
```

```
 3 del -Recurse -WhatIf "C:\Program Files"
 4 del -Recurse -WhatIf "C:\temp\*.*"
 5 del -WhatIf "C:\temp\*.*"
 6 Get-Alias del
```

По умолчанию отображаются последние 32 команды со своими порядковыми номерам (колонка `Id`). Число выводимых команд можно изменить с помощью параметра `-Count`. Например, следующая команда выведет три последних команды:

```
PS C:\> Get-History -Count 3

  Id CommandLine
  -- -----------
   5 del -WhatIf "C:\temp\*.*"
   6 Get-Alias del
   7 Get-History
```

Можно выделять из журнала сеанса команды, удовлетворяющие определенному критерию. Для этого используется процедура конвейеризации объектов и специальный командлет `Where-Object`, который подробно обсуждается в *главе 5*. Например, для вывода всех команд, содержащих слово `del`, можно выполнить следующую команду:

```
PS C:\> Get-History | Where-Object {$_.CommandLine -like "*del*"}

  Id CommandLine
  -- -----------
   3 del -Recurse -WhatIf "C:\Program Files"
   4 del -Recurse -WhatIf "C:\temp\*.*"
   5 del -WhatIf "C:\temp\*.*"
   6 Get-Alias del
```

Полученный с помощью `Get-History` список команд можно экспортировать во внешний файл в формате XML или CSV (текстовый файл с запятыми в качестве разделителя). Для этого вновь нужно применять конвейеризацию команд и специальные командлеты для экспорта данных в определенный формат. Например, следующая команда сохраняет журнал команд в CSV-файле c:\history.csv:

```
PS C:\> Get-History | Export-Csv c:\history.csv
```

С помощью командлета `Add-History` можно добавлять команды в журнал сеанса (например, для создания журнала, содержащего команды, введенные

за несколько сеансов работы). Например, следующая команда добавит в журнал сеанса команды, сохраненные в файле c:\history.csv:

```
PS C:\> Import-Csv c:\history.csv | Add-History
```

Командлет `Invoke-History` (псевдонимы `r`, сокращение от "repeat" или "rerun", и `ihy`) позволяет повторно выполнять команды из журнала сеанса, при этом команды можно задавать по их порядковому номеру или первым символам, а также получать по конвейеру от командлета `Get-History`. Приведем несколько примеров.

Вызов последней команды:

```
PS C:\> Invoke-History
```

Вызов третьей команды по ее порядковому номеру:

```
PS C:\> Invoke-History -Id 3
```

или просто

```
PS C:\> r 3
```

Вызов последней команды `Get-Help`:

```
PS C:\> Invoke-History Get-He
```

Вызов команд, полученных по конвейеру от командлета `Get-History`:

```
PS C:\> Get-History | Where-Object {$_.commandLine -like "*del*"} |
Invoke-History
```

Протоколирование действий в сеансе работы

В предыдущем разделе мы научились вызывать список команд, выполненных во время сеанса работы, и сохранять этот список во внешний файл. В оболочке PowerShell также можно записывать в текстовый файл не только запускаемые команды, но и результат их выполнения, то есть сохранять в файле весь сеанс работы или какую-либо его часть. Для этой цели служит командлет `Start-Transcript`, имеющий следующий синтаксис (указаны только основные параметры):

```
Start-Transcript [[-Path] <строка>] [-NoClobber] [-Append]
```

Параметр `Path` здесь задает путь к текстовому файлу с протоколом работы (путь должен быть указан явно, подстановочные знаки не поддерживаются). Давайте запустим протоколирование работы с сохранением протокола в файле c:\transcript.txt:

```
PS C:\> Start-Transcript -Path c:\transcript.txt
Транскрибирование запущено, выходной файл c:\transcript.txt
```

Выполним теперь какую-нибудь команду и завершим протоколирование с помощью командлета Stop-Transcript:

```
PS C:\> Get-Help Get-Process
ИМЯ
    Get-Process

ОПИСАНИЕ
    Отображает процессы, выполняющиеся на локальном компьютере.
. . .

PS C:\> Stop-Transcript
Транскрибирование остановлено, выходной файл c:\transcript.txt
```

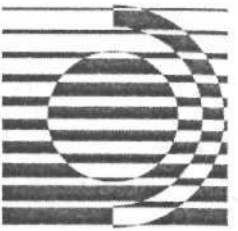
Посмотрим на содержимое файла c:\transcript.txt:

```
PS C:\> type c:\transcript.txt
**********************
Windows PowerShell Начало записи протокола
Время запуска: 20080211160911
Имя пользователя : POPOV\User
Компьютер        : POPOV (Microsoft Windows NT 5.1.2600 Service Pack 2)
**********************
Транскрибирование запущено, выходной файл c:\transcript.txt
PS C:\> get-help get-process

ИМЯ
    Get-Process

ОПИСАНИЕ
    Отображает процессы, выполняющиеся на локальном компьютере.

. . .

PS C:\> Stop-Transcript
**********************
Windows PowerShell Конец записи протокола
Время окончания: 20080211161213
**********************
```

Как видите, в файле протокола дополнительно записаны информация о дате и времени начала протоколирования и его завершения, а также об имени компьютера и учетной записи пользователя, запускавшего протоколирование.

Если имя для файла протокола не указано, то он будет сохраняться в файле, путь к которому задан в значении глобальной переменной `$Transcript`. Если эта переменная не определена, то командлет `Start-Transcript` сохраняет протоколы в каталоге "$Home\Мои документы" в файлах "PowerShell_transcript.<метка-времени>.txt" (в переменной `$Home` хранится путь к домашнему каталогу работающего пользователя).

Если файл протокола, в который должен начать сохраняться сеанс работы, уже существует, то по умолчанию он будет переписан. Параметр `-Append` командлета `Start-Transcript` включает режим добавления нового протокола к существующему файлу. Если же указан параметр `-NoClobber`, а файл протокола уже существует, то командлет `Start-Transcript` не выполняет никаких действий.



Глава 4

Настройка оболочки

В *главе 2* мы рассмотрели, каким образом можно настроить некоторые аспекты оболочки PowerShell под свои требования. Мы научились создавать собственные псевдонимы и функции для часто используемых командлетов или других команд, определять собственные переменные и диски.

Теперь мы займемся настройкой внешнего вида оболочки, а также научимся сохранять свои настройки и автоматически загружать их в каждом сеансе работы в PowerShell.

Настройка ярлыка PowerShell

Как уже упоминалось ранее, PowerShell функционирует в обычном консольном окне Windows, в котором работает и командный интерпретатор cmd.exe. Поэтому если вы настраивали окно cmd.exe, то никаких сложностей с настройкой размера, положения, используемых шрифтов и прочих параметров окна PowerShell возникнуть не должно.

Для настройки параметров нужно запустить оболочку PowerShell, щелкнуть правой кнопкой мыши, заголовке окна (или любой кнопкой мыши на пиктограмме окна) и выбрать пункт **Свойства** в появившемся контекстном меню. В результате будет открыто диалоговое окно для изменения настроек командного окна PowerShell (рис. 4.1).

Это окно имеет несколько вкладок (**Общие**, **Шрифт**, **Расположение** и **Цвета**), на которых сгруппированы элементы управления, позволяющие управлять соответствующими категориями параметров. Подробно рассматривать эти параметры мы не будем, назначение их вполне понятно.

Сделав все нужные изменения, нужно нажать кнопку **OK**, после чего система выдаст диалоговое окно **Изменение свойств ярлыка** (рис. 4.2).

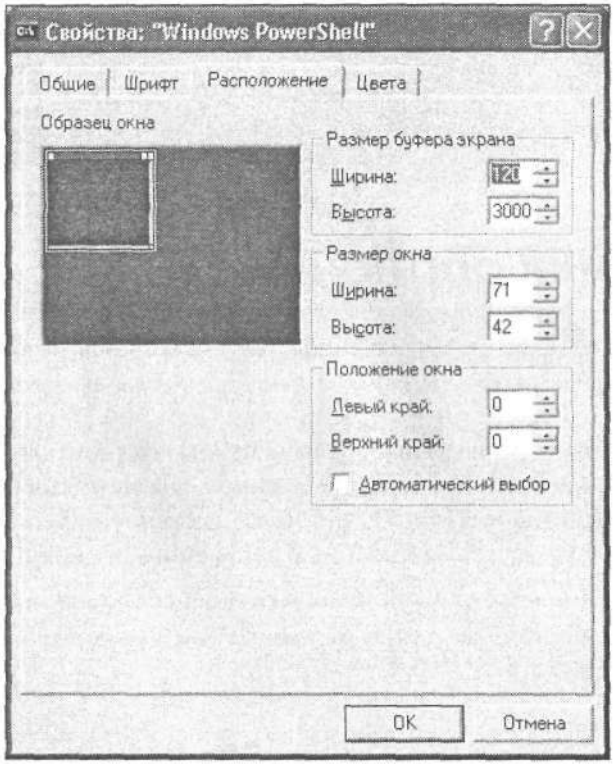


Рис. 4.1. *Диалоговое окно для настройки параметров оболочки* PowerShell

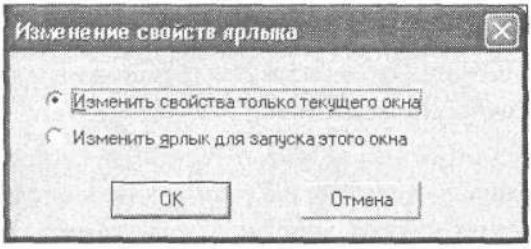


Рис. 4.2. Выбор объекта для применения изменений в настройках

Если вы хотите, чтобы сделанные изменения действовали постоянно (при каждом открытии нового командного окна PowerShell), то нужно выбрать *переключатель* ***Изменить ярлык для запуска этого окна***. Если же выбрать переключатель **Изменить свойства только текущего окна**, то настройки будут применены к активному окну PowerShell и будут действовать до окончания сеанса работы в данном окне.

Программное изменение свойств консоли PowerShell

Система PowerShell позволяет настраивать различные параметры командного окна (размер, цвет текста и фона, вид приглашения и т. п.) непосредственно из оболочки. Для этого можно воспользоваться командлетом `Get-Host`, который по умолчанию выводит информацию о самой оболочке PowerShell (версия, региональные настройки и т. д.):

```
PS C:\> Get-Host

Name             : ConsoleHost
Version          : 1.0.0.0
InstanceId       : 8560aa9f-1063-4005-9b4e-ae9649286b3a
UI               : System.Management.Automation.Internal.Host...
CurrentCulture   : ru-RU
CurrentUICulture : ru-RU
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColor...
```

Нам понадобится свойство `UI` (это объект .NET-класса `System.Management.Automation.Internal.Host.InternalHostUserInterface`). В свою очередь объект `UI` имеет свойство `RawUI`, позволяющее получить доступ к параметрам командного окна PowerShell. Для просмотра данных параметров выполним следующую команду:

```
PS C:\> (Get-Host).UI.RawUI

ForegroundColor       : DarkYellow
BackgroundColor       : DarkMagenta
CursorPosition        : 0,18
WindowPosition        : 0,0
CursorSize            : 25
BufferSize            : 120,3000
WindowSize            : 120,50
MaxWindowSize         : 120,80
MaxPhysicalWindowSize : 160,80
KeyAvailable          : False
WindowTitle           : Windows PowerShell
```

Замечание

В последней команде командлет `Get-Host` был заключен в круглые скобки. Это означает, что система вначале запускает данный командлет и получает

выходной объект в результате его работы. Затем извлекается свойство UI данного объекта (объект UI) и свойство RawUI объекта UI. На экран окончательно выводятся значения свойств объекта RawUI.

Многие свойства объекта RawUI можно изменять, настраивая тем самым соответствующие свойства командного окна.

Цвета текста и фона

За цвет текста в командном окне PowerShell отвечает свойство ForegroundColor объекта RawUI, а за цвет фона — свойство BackgroundColor. Для изменения данных свойств удобнее предварительно сохранить объект RawUI в отдельную переменную:

```
PS C:\> $a=(get-host).UI.RawUI
```

Теперь можно присвоить свойствам новые значения (PowerShell поддерживает 16 цветов, которым соответствуют следующие символьные константы: Black, Gray, Red, Magenta, Yellow, Blue, Green, Cyan, White, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, DarkGray и DarkBlue). Установим темно-желтый текст на синем фоне:

```
PS C:\> $a.BackgroundColor = "DarkBlue"
PS C:\> $a.ForegroundColor = "Yellow"
```

Замечание

В оболочке cmd.exe цвет текста и фона настраивался с помощью команды color, при этом изменения действовали на все командное окно, включая строки, введенные ранее. В PowerShell цвета меняются только у строк, которые вводятся после команд, меняющих свойство Background и/или Foreground; а введенные ранее строки остаются без изменений.

Заголовок командного окна

По умолчанию в заголовке командного окна отображается строка "Windows PowerShell". Для изменения этого заголовка нужно записать новое значение в свойство WindowTitle объекта RawUI:

```
PS C:\> $a=(get-host).UI.RawUI
PS C:\> $a.WindowTitle="Мое командное окно"
```

Сразу после выполнения последней команды заголовком окна PowerShell станет строка "Мое командное окно" (рис. 4.3).

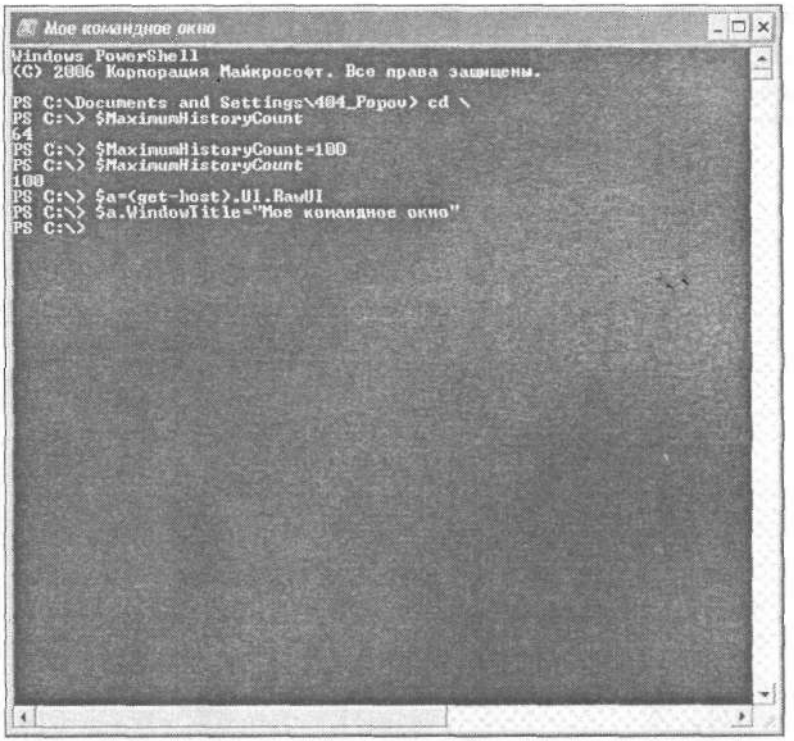


Рис. 4.3. Измененный заголовок командного окна PowerShell

Размеры командного окна

Посмотрим еще раз на объект `RawUI`:

```
PS C:\> (Get-Host).UI.RawUI

ForegroundColor       : Yellow
BackgroundColor       : DarkBlue
CursorPosition        : 0,28
WindowPosition        : 0,0
CursorSize            : 25
BufferSize            : 120,3000
WindowSize            : 120,50
MaxWindowSize         : 120,59
MaxPhysicalWindowSize : 146,59
KeyAvailable          : False
WindowTitle           : Мое командное окно
```

Размеры окна PowerShell хранятся в свойстве `WindowSize` объекта `RawUI` в виде пары чисел: количество символов в строке и количество строк в окне. На самом деле свойство WindowSize — это тоже объект, имеющий свойства `Width` и `Height`:

```
PS C:\> (Get-Host).UI.RawUI.WindowSize

Width             Height
-----             ------
  120                 50
```

Поэтому изменить размеры командного окна PowerShell будет несколько сложнее, чем цвета текста и фона или заголовок окна. Сначала мы, как обычно, сохраним в переменной `$a` объект `RawUI`:

```
PS C:\> $a=(get-host).UI.RawUI
```

Затем в переменной `$b` сохраним объект `WindowSize`:

```
PS C:\> $b=$a.WindowSize
```

Изменим значения свойств `Width` и `Height` переменной `$b`:

```
PS C:\> $b.Width=80
PS C:\> $b.Height=25
```

Отметим, что при этом размеры окна PowerShell еще не изменились. Присвоим теперь переменную `$b` в качестве значения свойству `WindowSize` переменной `$a`:

```
PS C:\> $a.WindowSize=$b
```

Сразу после выполнения данной команды размеры окна PowerShell изменятся.

Приглашение командной строки

Перейдем теперь к настройке приглашения командной строки. Напомним, что в оболочке cmd.exe за вид приглашения командной строки отвечает специальная переменная среды `PROMPT`. Посмотреть и изменить ее значение можно с помощью команды `set` (данную команду нужно выполнять в командной строке cmd.exe, а не PowerShell):

```
C:\> set prompt
PROMPT=$P$G
```

В оболочке PowerShell приглашение командной строки контролируется с помощью функции `Prompt`, которая должна возвращать одну строку. Посмотрим, не вдаваясь пока в подробности, как описана данная функция по умолчанию (она отображает символы `PS`, затем путь к текущему каталогу и символ >). Для этого можно выполнить следующую команду:

```
PS C:\> (Get-Item Function:Prompt).Definition
'PS ' + $(Get-Location) + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
```

Эквивалентом стандартного приглашения оболочки cmd.exe (путь к текущему каталогу и символ >) будет являться следующая функция `Prompt`:

```
PS C:\> Function Prompt{"$(Get-Location)> "}
C:\>
```

Как видите, вид приглашения меняется сразу после задания нового содержимого функции `Prompt`.

Замечание

При определении функции `Prompt` мы внутри строки, заключенной в двойные кавычки, использовали конструкцию `$(...)`, которая называется *подвыражением* (subexpression). Подвыражение — это блок кода на языке PowerShell, который в строке заменяется значением, полученным в результате выполнения этого кода.

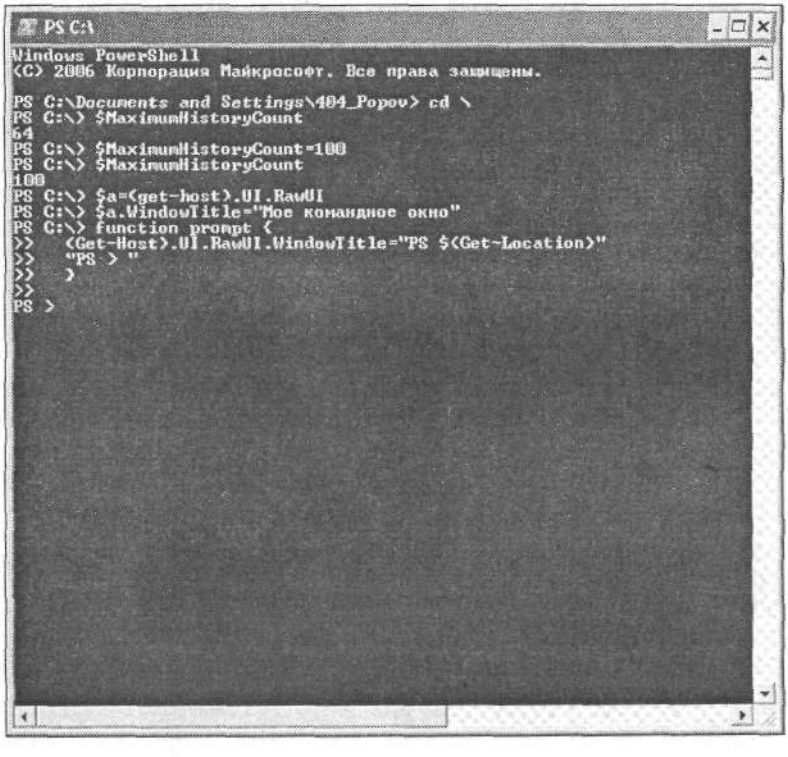


Рис. 4.4. Окно PowerShell, в заголовке которого указан путь к текущему каталогу

В функции `Prompt` можно не только формировать приглашение командной строки, но и выполнять любые другие действия. Например, с помощью функции `Prompt` можно решить проблему отображения длинных путей к те-

кущему каталогу (такие пути при отображении в приглашении командной строки занимают много места, и работать становится неудобно). Если определить функцию `Prompt` так, как в листинге 4.1, путь к текущему каталогу будет отображаться не в приглашении командной строки, а в заголовке командного окна (рис. 4.4).

Листинг 4.1. Вывод на экран всех параметров сценария

```
Function Prompt {
  (Get-Host).UI.RawUI.WindowTitle="PS $(Get-Location)"
  "PS > "
  }
```

Настройка пользовательских профилей

Итак, прочитав предыдущие разделы, вы настроили оболочку PowerShell для максимального удобства работы в ней: изменили цвета, поменяли приглашение командной строки, создали собственные псевдонимы и диски. Однако все эти настройки и изменения будут действовать только во время текущего сеанса работы и утратят силу после выхода из оболочки.

Для сохранения изменений необходимо создать так называемый *профиль* PowerShell и записать в него все команды, которые определяют нужные вам псевдонимы, функции, переменные и т. п. Профиль — это сценарий, который будет загружаться при каждом запуске PowerShell (наподобие командных файлов autoexec.bat или autoexec.nt в оболочке cmd.exe). Грамотно созданный профиль может упростить работу в PowerShell и администрирование операционной системы.

> ***Замечание***
>
> Создание и распространение профилей может помочь поддерживать единородное и согласованное представление оболочки PowerShell на нескольких компьютерах.

В PowerShell поддерживаются профили четырех типов. Дело в том, что модель PowerShell предусматривает возможность использования разных интерфейсов для языка (так называемых хостов) и консольное приложение powershell.exe представляет собой в сущности лишь один из таких хостов.

> ***Замечание***
>
> В настоящее время доступны несколько альтернативных хостов PowerShell, например PowerShell Plus и PowerShell Analyzer, которые можно загрузить с сайта **http://www.powershell.com**.

Итак, работая в оболочке powershell.exe, каждый пользователь может иметь четыре разных профиля:

- %windir%\system32\WindowsPowerShell\v1.0\profile.ps1 (напомним, что в переменной среды %windir% хранится путь к корневому каталогу Windows). Данный профиль действует на всех пользователей и на все оболочки (хосты);
- %windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1. Действие этого профиля распространяется на всех пользователей, использующих хост powershell.exe (Microsoft.PowerShell — это идентификатор для powershell.exe).
- "%UserProfile%\Мои документы\WindowsPowerShell\profile.ps1". Действие данного профиля распространяется только на текущего пользователя и на все хосты.
- "%UserProfile%\Мои документы\WindowsPowerShell\Microsoft.PowerShell_profile.ps1". Действие этого профиля распространяется только на текущего пользователя и только на хост powershell.exe.

> ***Замечание***
>
> Если на жестком диске имеются несколько профилей, которые могут быть загружены в конкретной ситуации, то предпочтение будет отдано наиболее узконаправленному.

Обычно при работе с оболочкой PowerShell используют профиль, специфичный для пользователя и оболочки, который называется *пользовательским профилем*. Данные о расположении этого профиля хранятся в специальной переменной `$profile`:

```
PS F:\> $profile
C:\Documents and Settings\404_Popov\Мои документы\WindowsPowerShell\
Microsoft.PowerShell_profile.ps1
```

Находясь в оболочке PowerShell, можно с помощью командлета `Test-Path` проверить, создан ли уже пользовательский профиль:

```
PS F:\> Test-Path $profile
False
```

Если профиль существует, эта команда возвратит `True`, в противном случае — `False`.

Для создания нового пользовательского профиля или изменения уже существующего нужно открыть в текстовом редакторе файл, путь к которому хранится в переменной `$profile`. Сделать это можно из Проводника Win-

кущему каталогу (такие пути при отображении в приглашении командной строки занимают много места, и работать становится неудобно). Если определить функцию `Prompt` так, как в листинге 4.1, путь к текущему каталогу будет отображаться не в приглашении командной строки, а в заголовке командного окна (рис. 4.4).

Листинг 4.1. Вывод на экран всех параметров сценария

```
Function Prompt {
  (Get-Host).UI.RawUI.WindowTitle="PS $(Get-Location)"
  "PS > "
  }
```

Настройка пользовательских профилей

Итак, прочитав предыдущие разделы, вы настроили оболочку PowerShell для максимального удобства работы в ней: изменили цвета, поменяли приглашение командной строки, создали собственные псевдонимы и диски. Однако все эти настройки и изменения будут действовать только во время текущего сеанса работы и утратят силу после выхода из оболочки.

Для сохранения изменений необходимо создать так называемый *профиль* PowerShell и записать в него все команды, которые определяют нужные вам псевдонимы, функции, переменные и т. п. Профиль — это сценарий, который будет загружаться при каждом запуске PowerShell (наподобие командных файлов autoexec.bat или autoexec.nt в оболочке cmd.exe). Грамотно созданный профиль может упростить работу в PowerShell и администрирование операционной системы.

> ***Замечание***
>
> *Создание и распространение профилей может помочь поддерживать единородное и согласованное представление оболочки PowerShell на нескольких компьютерах.*

В PowerShell поддерживаются профили четырех типов. Дело в том, что модель PowerShell предусматривает возможность использования разных интерфейсов для языка (так называемых хостов) и консольное приложение powershell.exe представляет собой в сущности лишь один из таких хостов.

> ***Замечание***
>
> В настоящее время доступны несколько альтернативных хостов PowerShell, например PowerShell Plus и PowerShell Analyzer, которые можно загрузить с сайта **http://www.powershell.com**.

Итак, работая в оболочке powershell.exe, каждый пользователь может иметь четыре разных профиля:

- %windir%\system32\WindowsPowerShell\v1.0\profile.ps1 (напомним, что в переменной среды %windir% хранится путь к корневому каталогу Windows). Данный профиль действует на всех пользователей и на все оболочки (хосты);
- %windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1. Действие этого профиля распространяется на всех пользователей, использующих хост powershell.exe (Microsoft.PowerShell — это идентификатор для powershell.exe).
- "%UserProfile%\Мои документы\WindowsPowerShell\profile.ps1". Действие данного профиля распространяется только на текущего пользователя и на все хосты.
- "%UserProfile%\Мои документы\WindowsPowerShell\Microsoft.PowerShell_profile.ps1". Действие этого профиля распространяется только на текущего пользователя и только на хост powershell.exe.

> ***ЗАМЕЧАНИЕ***
>
> Если на жестком диске имеются несколько профилей, которые могут быть загружены в конкретной ситуации, то предпочтение будет отдано наиболее узконаправленному.

Обычно при работе с оболочкой PowerShell используют профиль, специфичный для пользователя и оболочки, который называется *пользовательским профилем*. Данные о расположении этого профиля хранятся в специальной переменной `$profile`:

```
PS F:\> $profile
C:\Documents and Settings\404_Popov\Мои документы\WindowsPowerShell\
Microsoft.PowerShell_profile.ps1
```

Находясь в оболочке PowerShell, можно с помощью командлета `Test-Path` проверить, создан ли уже пользовательский профиль:

```
PS F:\> Test-Path $profile
False
```

Если профиль существует, эта команда возвратит `True`, в противном случае — `False`.

Для создания нового пользовательского профиля или изменения уже существующего нужно открыть в текстовом редакторе файл, путь к которому хранится в переменной `$profile`. Сделать это можно из Проводника Win-

dows или непосредственно из оболочки PowerShell с помощью следующей команды:

```
PS F:\> notepad $profile
```

Если файл с профилем уже существовал на диске, то в результате выполнения последней команды будет открыт Блокнот Windows с содержимым файла Microsoft.PowerShell_profile.ps1. Если же профиль еще не был создан, то будет выдано диалоговое окно с сообщением о том, что открываемый файл не найден. Нажав кнопку **OK** в этом окне, мы попадем в пустое окно Блокнота Windows. Теперь нужно ввести команды, которые мы хотим выполнять при каждой загрузке PowerShell (например, функцию `Prompt` из листинга 4.1), и сохранить файл под именем Microsoft.PowerShell_profile.ps1 в каталоге "%UserProfile%\Мои документы\WindowsPowerShell". При этом в диалоговом окне **Сохранить как** в поле **Тип файла** нужно выбрать пункт **Все файлы**, так как в противном случае к имени сохраняемого файла будет добавлено расширение txt и профиль не будет загружаться при старте оболочки (рис. 4.5).

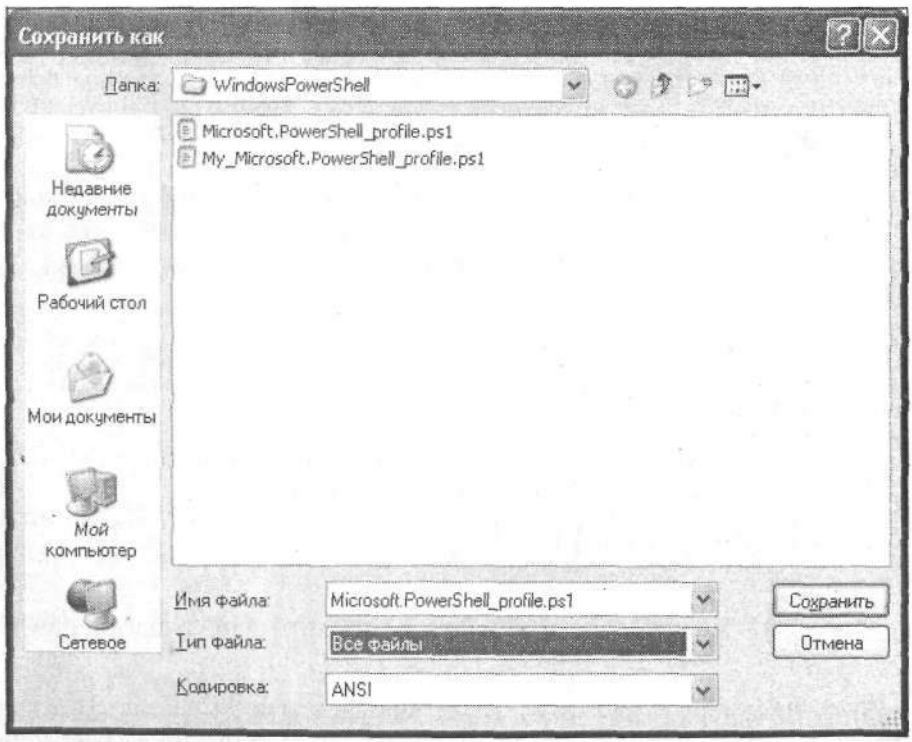


Рис. 4.5. Сохранение файла с пользовательским профилем

После сохранения файла с профилем можно проверить его содержимое из оболочки:

```
PS F:\> type $profile
Function Prompt {
  (Get-Host).UI.RawUI.WindowTitle="PS $(Get-Location)"
  "PS > "
  }
```

Итак, мы имеем пользовательский профиль, который будет загружаться при каждом старте оболочки PowerShell и изменять заголовок окна и приглашение командной строки.

Завершим текущий сеанс работы и заново запустим PowerShell. Скорее всего, мы получим сообщение об ошибке наподобие следующего:

```
Не удается загрузить файл C:\Documents and Settings\404_Popov\Мои
документы\WindowsPowerShell\Microsoft.PowerShell_profile.ps1,
так как выполнение сценариев запрещено для данной системы.
Введите "get-help about_signing" для получения дополнительных сведений.
В строка:1 знак:2
+ .  <<<< 'C:\Documents and Settings\404_Popov\Мои документы\
WindowsPowerShell\Microsoft.PowerShell_profile.ps1'
```

Дело в том, что для повышения уровня безопасности по умолчанию в PowerShell действует *политика выполнения*, которая запрещает загрузку профилей и вообще выполнение любых сценариев (разрешается только выполнять команды в интерактивном режиме). Поэтому нам нужно научиться настраивать политику выполнения таким образом, чтобы сценарии PowerShell начали запускаться.

Политики выполнения сценариев

Политика выполнения (execution policy) оболочки PowerShell определяет, можно ли на данном компьютере выполнять сценарии PowerShell (в том числе загружать пользовательские профили), и если да, должны ли они быть подписаны цифровой подписью. Политика выполнения PowerShell хранится в реестре операционной системы и не удаляется даже при переустановке оболочки PowerShell.

Возможные политики выполнения PowerShell описаны в табл. 4.1 (получить аналогичную информацию в PowerShell можно с помощью команды `Get-Help about_signing`).

Таблица 4.1. *Политики выполнения PowerShell*

Название политики	Описание
`Restricted`	Данная политика выполнения используется по умолчанию, она запрещает выполнение сценариев и загрузку профилей (можно пользоваться только одиночными командами PowerShell в интерактивном режиме)
`AllSigned`	Выполнение сценариев PowerShell разрешено, однако все сценарии (как загруженные из Интернета, так и локальные) должны иметь цифровую подпись надежного издателя. Перед выполнением сценариев надежных издателей запрашивается подтверждение
`RemoteSigned`	Выполнение сценариев PowerShell разрешено, при этом все сценарии и профили, загруженные из Интернета (в том числе по электронной почте и с помощью программ мгновенного обмена сообщениями), должны иметь цифровую подпись надежного издателя, а локальные сценарии могут быть неподписанными. При запуске сценариев надежных издателей подтверждение не запрашивается
`Unrestricted`	Разрешается выполнение любых сценариев PowerShell без проверки цифровой подписи. При запуске сценариев и профилей, загруженных из Интернета, выдается предупреждение

Узнать, какая политика выполнения является активной, можно с помощью командлета `Get-ExecutionPolicy`:

```
PS C:\> Get-ExecutionPolicy
Restricted
```

Как видите, по умолчанию действует политика `Restricted`, запрещающая запуск любых сценариев PowerShell, включая пользовательские профили.

Командлет `Set-ExecutionPolicy` позволяет сменить политику выполнения. Например, для установки политики выполнения `RemoteSigned` нужно выполнить следующую команду:

```
Set-ExecutionPolicy RemoteSigned
```

Замечание

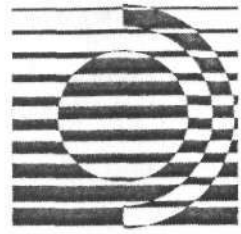
Не забывайте о функции автоматического завершения команд — нет необходимости запоминать названия возможных политик, можно просто нажимать

клавишу <Tab> после имени командлета, и система будет подставлять различные варианты политик.

Проверим снова текущую политику:

```
PS C:\> Get-ExecutionPolicy
RemoteSigned
```

Теперь завершим сеанс работы в PowerShell и вновь запустим оболочку. На этот раз никаких сообщений об ошибке выдаваться не должно, функция `Prompt` из пользовательского профиля должна отработать корректно, и в заголовке командного окна PowerShell отобразится путь к текущему каталогу (см. рис. 4.4).



Глава 5

Работа с объектами

В предыдущих главах мы уже неоднократно говорили о том, что все действия в оболочке PowerShell связаны с операциями над объектами.

Нелишним будет напомнить, что объект — это совокупность данных (*свойства* объекта) и способов работы с этими данными (*методы* объекта). Конкретная структура объекта (состав свойств и методов) задается его *типом* (например, любому файлу на жестком диске соответствует объект типа `FileInfo`). Набор типов, использующихся в PowerShell, базируется на типах унифицированной платформы .NET Framework, повсеместно используемой в современных версиях операционной системы Windows.

Свойство объекта — это сведения о его состоянии или параметрах. Например, у объекта `FileInfo` имеется свойство `Length` (длина), соответствующее размеру файла, который представлен данным объектом.

Метод объекта является действием, которое можно совершать над элементом, представленным данным объектом. Например, у объекта `FileInfo` имеется метод `CopyTo`, с помощью которого можно скопировать файл (при вызове этого метода происходит копирование представленного объектом файла на уровне файловой системы).

Свойства и методы объектов используются в командлетах PowerShell для выполнения различных действий и работы с данными. При этом в PowerShell поддерживается заимствованный из других интерфейсов командной строки механизм *конвейеризации* или *композиции* команд, что значительно повышает эффективность работы.

Конвейеризация объектов в PowerShell

В большинстве оболочек командной строки, включая cmd.exe, под конвейеризацией понимается объединение (композиция) нескольких команд путем последовательного перенаправления выходного потока одной команды

во входной поток другой, что позволяет передавать текстовую информацию между разными процессами.

Механизм композиции команд представляет собой, вероятно, наиболее ценную концепцию, используемую в интерфейсах командной строки. Конвейеры не только снижают усилия, прилагаемые при вводе сложных команд, но и облегчают отслеживание выполняемых командами действий. Полезной чертой конвейеров является то, что они не зависят от числа передаваемых элементов, так как конвейер действует на каждый элемент отдельно. Кроме того, каждая команда в конвейере (называемая элементом конвейера) обычно передает свой вывод следующей команде в конвейере, элемент за элементом. Благодаря этому, как правило, снижается потребление ресурсов для сложных команд и появляется возможность получать выводимую информацию немедленно.

В оболочке PowerShell также очень широко используется механизм конвейеризации команд, однако здесь по конвейеру передается не поток текста, как во всех других оболочках, а объекты. При этом с элементами конвейера можно производить различные манипуляции: фильтровать объекты по определенному критерию, сортировать и группировать объекты, изменять их структуру.

Конвейер в PowerShell — это последовательность команд, разделенных между собой знаком | (вертикальная черта). Каждая команда в конвейере получает объект от предыдущей команды, выполняет определенные операции над ним и передает следующей команде в конвейере. С точки зрения пользователя, объекты упаковывают связанную информацию в форму, в которой информацией проще манипулировать как единым блоком и из которой при необходимости извлекаются определенные элементы.

Передача данных между командами в виде объектов имеет большое преимущество над обычным обменом информацией посредством потока текста. Ведь команда, принимающая поток текста от другой утилиты, должна его проанализировать, разобрать и выделить нужную ей информацию, а это может быть непросто, так как обычно вывод команды больше ориентирован на визуальное восприятие человеком (это естественно для интерактивного режима работы), а не на удобство последующего синтаксического разбора.

При передаче по конвейеру объектов этой проблемы не возникает, здесь нужная информация извлекается из элемента конвейера простым обращением к соответствующему свойству объекта. Однако теперь возникает новый вопрос: как можно узнать, какие именно свойства есть у объектов, передаваемых по конвейеру? Ведь при выполнении того или иного командлета мы на экране видим только одну или несколько колонок отформатированного

текста. Например, запустим командлет Get-Process, который выводит информацию о запущенных в системе процессах:

```
PS C:\> Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    158      11    45644      22084   126   159.69   2072 AcroRd32
     98       5     1104        284    32     0.10    256 alg
     39       1      364        364    17     0.26   1632 ati2evxx
     57       3     1028        328    30     0.38    804 atiptaxx
    434       6     2548       3680    27    21.96    800 csrss
     64       3      812        604    29     0.22   1056 ctfmon
    364      11    14120       9544    69    11.82    456 explorer
     24       2     1532       2040    29     5.34   2532 Far
. . .
```

Фактически на экране мы видим только сводную информацию (результат форматирования полученных данных), а не полное представление выходного объекта. Из этой информации непонятно, сколько точно свойств имеется у объектов, генерируемых командой Get-Process, и какие имена имеют эти свойства. Например, мы хотим найти все "зависшие" процессы, которые не отвечают на запросы системы. Можно ли это сделать с помощью командлета Get-Process, какое именно свойство для этого нужно проверять у выводимых объектов?

Для ответа на подобные вопросы нужно, прежде всего, научиться исследовать структуру объектов PowerShell, узнавать, какие свойства и методы имеются у этих объектов.

Просмотр структуры объектов (командлет *Get-Member*)

Для анализа структуры объекта, возвращаемого определенной командой, проще всего направить этот объект по конвейеру на командлет Get-Member (псевдоним gm), например:

```
PS C:\> Get-Process | Get-Member

   TypeName: System.Diagnostics.Process

Name         MemberType     Definition
----         ----------     ----------
Handles      AliasProperty  Handles = Handlecount
```

```
Name            AliasProperty  Name = ProcessName
NPM             AliasProperty  NPM = NonpagedSystemMemorySize
PM              AliasProperty  PM = PagedMemorySize
VM              AliasProperty  VM = VirtualMemorySize
WS              AliasProperty  WS = WorkingSet
. . .
Responding      Property       System.Boolean Responding {get;}
. . .
```

В результате на экране мы видим, какой .NET-тип имеют объекты, возвращаемые в ходе работы исследуемого командлета (в нашем примере это тип `System.Diagnostic.Process`), а также полный список элементов объекта (в частности, интересующее нас свойство `Responding`, определяющее "зависшие" процессы). При этом на экран выводится очень много элементов разных типов (имена и псевдоним свойств, имена методов и т. д.), и такой длинный список становится неудобно просматривать. Командлет `Get-Member` имеет параметр `-MemberType`, позволяющий перечислить только элементы объекта определенного типа. Например, для вывода только элементов объекта, являющихся свойствами этого объекта, используется параметр `-MemberType` со значением `Property`:

```
PS C:\> Get-Process | Get-Member -MemberType Property

   TypeName: System.Diagnostics.Process

Name                   MemberType Definition
----                   ---------- ----------
BasePriority           Property   System.Int32 BasePriority {get;}
Container              Property   System.ComponentModel.IContainer...
EnableRaisingEvents    Property   System.Boolean EnableRaisingEven...
ExitCode               Property   System.Int32 ExitCode {get;}
ExitTime               Property   System.DateTime ExitTime {get;}
Handle                 Property   System.IntPtr Handle {get;}
HandleCount            Property   System.Int32 HandleCount {get;}
HasExited              Property   System.Boolean HasExited {get;}
Id                     Property   System.Int32 Id {get;}
MachineName            Property   System.String MachineName {get;}
. . .
Responding             Property   System.Boolean Responding {get;}
. . .
```

Как видите, процессам операционной системы соответствуют объекты, имеющие очень много свойств, на экран же при работе командлета `Get-Process` выводятся лишь несколько из них. На самом деле способы отображения в оболочке PowerShell объектов различных типов задаются несколькими конфигурационными файлами с расширением ps1xml в формате XML, которые находятся в каталоге, где установлен файл powershell.exe (путь к этому каталогу хранится в переменной `$PSHome`). Список этих файлов можно получить с помощью следующей команды:

```
PS C:\> dir $pshome\*format*.ps1xml

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS.1\system32\
WindowsPowerShell\v1.0

Mode          LastWriteTime  Length Name
----          -------------  ------ ----
-----  08.09.2006     12:28   22120 certificate.format.ps1xml
-----  08.09.2006     12:28   60703 dotnettypes.format.ps1xml
-----  08.09.2006     12:28   19730 filesystem.format.ps1xml
-----  08.09.2006     12:28  250197 help.format.ps1xml
-----  08.09.2006     12:28   65283 powershellcore.format.ps1xml
-----  08.09.2006      1:28   13394 powershelltrace.format.ps1xml
-----  08.09.2006     12:28   13540 registry.format.ps1xml
```

В частности, правило форматирования объекта типа `System.Diagnostic.Process` находится в файле `dotnettypes.format.ps1xml`. Напрямую редактировать конфигурационные файлы не рекомендуется, а в случае необходимости можно создать собственные файлы форматирования и с помощью командлета `Update-FormatData` включить их в состав автоматически загружаемых файлов.

Теперь, когда мы знаем, какие свойства имеют объекты, передаваемые по конвейеру, перейдем к рассмотрению возможных операций над элементами конвейера.

Фильтрация объектов (командлет *Where-Object*)

В PowerShell поддерживается возможность *фильтрации объектов* в конвейере, то есть удаления из конвейера объектов, не удовлетворяющих определенному условию. Данную функциональность обеспечивает командлет `Where-Object`, позволяющий проверить каждый объект, проходящий через конвейер,

и передать его дальше по конвейеру лишь в том случае, если объект удовлетворяет условиям проверки.

Условие проверки в `Where-Object` задается в виде *блока сценария* (scriptblock) — одной или нескольких команд PowerShell, заключенных в фигурные скобки `{}`. Блок сценария указывается после имени командлета `Where-Object`. Результатом выполнения блока сценария в командлете `Where-Object` должно быть значение логического типа: `$True` (истина, в этом случае объект проходит далее по конвейеру) или `$False` (ложь, в этом случае объект далее по конвейеру не передается).

Например, для вывода информации об остановленных службах в системе (объекты, возвращаемые командлетом `Get-Service`, у которых свойство `Status` равно `"Stopped"`) можно использовать следующий конвейер:

```
PS C:\> Get-Service | Where-Object {$_.Status -eq "Stopped"}

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Оповещатель
Stopped  AppMgmt            Управление приложениями
Stopped  aspnet_state       ASP.NET State Service
Stopped  cisvc              Служба индексирования
Stopped  ClipSrv            Сервер папки обмена
. . .
```

Другой пример — оставим в конвейере только те процессы, у которых значение идентификатора (свойство `Id`) больше 1000:

```
PS C:\> Get-Process | Where-Object {$_.Id -gt 1000}

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    158       9    37768      26620   125    28.49   1752 AcroRd32
     39       1      364        420    17     0.16   1632 ati2evxx
     57       3     1028        804    30     0.40   1988 atiptaxx
     24       2     1460       1052    29     0.62   2084 Far
    720      75    38516      10508   153    50.96   1756 kavsvc
     36       2      728         32    23     0.06   1792 klswd
     33       2      792       1984    25     1.15   1412 notepad
    242     158    30544       5780   180     7.96   1784 outpost
    252       5    34384      27192   137     5.15   2904 powershell
    143       5     3252       1028    42     0.31   1528 spoolsv
```

```
194     5     2928     1340    59     0.34   1040 svchost
301    14     1784     1316    37     0.80   1116 svchost
1647   65    20820    11548   101    13.21   1152 svchost
55      3     1088      724    27     0.46   1224 svchost
170     6     1568      960    35     0.14   1604 svchost
120     4     2356     1292    35     0.22   1876 svchost
22      2      504      584    23     0.06   1764 winampa
430    13     8472    11352   246    57.63   3216 WINWORD
154     4     2112     2104    38    28.13   2032 wmiprvse
```

В блоках сценариев командлета `Where-Object` для обращения к текущему объекту конвейера и извлечения нужных свойств этого объекта используется специальная переменная `$_`, которая создается оболочкой PowerShell автоматически.

> ***Замечание***
>
> Данная переменная используется и в других командлетах, производящих обработку элементов конвейера.

Как можно понять из примеров, в блоке сценария используются специальные *операторы сравнения*. Основные операторы сравнения приведены в табл. 5.1.

> ***Замечание***
>
> В PowerShell для операторов сравнения не используются обычные символы > или <, так как в командной строке они обычно означают перенаправление ввода/вывода.

Таблица 5.1. *Операторы сравнения в PowerShell*

Оператор	Значение	Пример (возвращается значение True)
`-eq`	равно	`10 -eq 10`
`-ne`	не равно	`9 -ne 10`
`-lt`	меньше	`3 -lt 4`
`-le`	меньше или равно	`3 -le 4`
`-gt`	больше	`4 -gt 3`
`-ge`	больше или равно	`4 -ge 3`
`-like`	сравнение на совпадение с учетом подстановочного знака во втором операнде	`"file.doc" -like "f*.doc"`

Таблица 5.1 (окончание)

Оператор	Значение	Пример (возвращается значение True)
-notlike	сравнение на несовпадение с учетом подстановочного знака во втором операнде	"file.doc" -notlike "f*.rtf"
-contains	содержит	1,2,3 -contains 1
-notcontains	не содержит	1,2,3 -notcontains 4

Операторы сравнения можно соединять друг с другом с помощью *логических операторов* (табл. 5.2).

Таблица 5.2. Логические операторы в PowerShell

Оператор	Значение	Пример (возвращается значение True)
-and	логическое И	(10 -eq 10) -and (1 -eq 1)
-or	логическое ИЛИ	(9 -ne 10) -or (3 -eq 4)
-not	логическое НЕ	-not (3 -gt 4)
!	логическое НЕ	!(3 -gt 4)

Замечание

Более подробно операторы сравнения и логические операторы рассматриваются в *главе 7*.

Сортировка объектов (командлет *Sort-Object*)

Сортировка элементов конвейера — еще одна операция, которая часто применяется при конвейерной обработке объектов. Данную операцию осуществляет командлет `Sort-Object`: ему передаются имена свойств, по которым нужно произвести сортировку объектов, проходящих по конвейеру, а он возвращает данные, упорядоченные по значениям этих свойств.

Например, для вывода списка запущенных в системе процессов, упорядоченного по затраченному процессорному времени (свойство `cpu`), можно воспользоваться следующим конвейером:

```
PS C:\> Get-Process | Sort-Object -Property cpu

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      0       0        0         16     0               0 Idle
```

```
    36     2      728       32    23     0.05   1792 klswd
    98     5     1104      764    32     0.09    252 alg
    21     1      164       60     4     0.09    748 smss
    39     1      364      464    17     0.12   1644 ati2evxx
   163     6     1536     1404    35     0.12   1612 svchost
    55     3     1088      852    27     0.14   1220 svchost
. . .
   398     5    36140    26408   137     9.97   1984 powershell
   375    12    15020    10456    75    14.03   1116 explorer
   376     0        0       36     2    14.97      4 System
   409     6     2500     3192    26    20.10    812 csrss
  1513    54    13528     9800    95    25.78   1156 svchost
   717    75    37432      704   145    56.97   1748 kavsvc
   152     4     2372     2716    38    58.09   2028 wmiprvse
   307    13    10952    27080   173  9128.03   1200 WINWORD
```

Параметр `-Property` в командлете `Sort-Object` используется по умолчанию, поэтому имя этого параметра можно не указывать. Для сортировки в обратном порядке используется параметр `-Descending`:

```
PS C:\> Get-Process | Sort-Object cpu -Descending
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    307     13    10956      27040   173  9152.23   1200 WINWORD
    152      4     2372       2716    38    59.19   2028 wmiprvse
    717     75    37432       1220   145    57.15   1748 kavsvc
   1524     54    13528       9800    95    26.13   1156 svchost
    410      6     2508       3224    26    20.62    812 csrss
    376      0        0         36     2    15.11      4 System
    377     13    15020      10464    75    14.20   1116 explorer
    374      5    36484      26828   137    10.53   1984 powershell
. . .
     55      3     1088        852    27     0.14   1220 svchost
     39      1      364        464    17     0.13   1644 ati2evxx
    163      6     1536       1404    35     0.12   1612 svchost
     21      1      164         60     4     0.09    748 smss
     98      5     1104        764    32     0.09    252 alg
     36      2      728         32    23     0.05   1792 klswd
      0      0        0         16     0               0 Idle
```

В рассмотренных нами примерах конвейеры состояли из двух командлетов. Это не обязательное условие, конвейер может объединять и большее количество команд, например:

```
PS C:\> Get-Process | Where-Object {$_.Id -gt 1000} | Sort-Object cpu
 -Descending

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    307      13    10956      27040   173  9152.23   1200 WINWORD
    152       4     2372       2716    38    59.19   2028 wmiprvse
    717      75    37432       1220   145    57.15   1748 kavsvc
   1524      54    13528       9800    95    26.13   1156 svchost
    377      13    15020      10464    75    14.20   1116 explorer
    374       5    36484      26828   137    10.53   1984 powershell
    149       4     2940       1108    41     9.34   1248 kav
    240     158    29536      10388   175     5.61   1780 outpost
    281      14     1764       1688    37     0.34   1120 svchost
    140       5     3208       1220    41     0.32   1524 spoolsv
     64       3      812       1080    29     0.30   1252 ctfmon
    193       5     2916       1488    59     0.29   1040 svchost
    120       4     2364       1228    35     0.26   1876 svchost
     55       3     1088        852    27     0.14   1220 svchost
     39       1      364        464    17     0.13   1644 ati2evxx
    163       6     1536       1404    35     0.12   1612 svchost
     36       2      728         32    23     0.05   1792 klswd
```

В результате выполнения последнего конвейера из трех командлетов мы получили упорядоченный по количеству затраченного процессорного времени список процессов, идентификатор которых больше 1000.

Выделение объектов и свойств (командлет *Select-Object*)

В PowerShell имеется командлет `Select-Object`, с помощью которого можно выделять указанное количество объектов с начала или с конца конвейера, выбирать уникальные объекты из конвейера, а также выделять определенные свойства в объектах, проходящих по конвейеру.

Для выделения из конвейера нескольких первых или последних объектов следует воспользоваться соответственно параметрами `-First` или `-Last`

командлета Select-Object. Например, следующий конвейер команд выведет на экран информацию о пяти процессах, использующих наибольший объем памяти:

```
PS C:\> Get-Process | Sort-Object WS | Select-Object -Last 5

Handles  NPM(K)    PM(K)    WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----    ----- -----   ------     -- -----------
    398      12    14736     8096    78    12.99    740 explorer
   1638      66    21368    12292   103    30.03   1152 svchost
    280      12    10252    14900   139   124.56   3216 WINWORD
    158       9    37776    19704   125    36.21   1752 AcroRd32
    297       6    38408    20844   137     8.53   2904 powershell
```

Разберем работу данного конвейера команд. Первый командлет в конвейере (Get-Process) возвращает массив объектов, соответствующих запущенным в системе процессам. Второй командлет Sort-Object упорядочивает проходящие по конвейеру объекты по значению свойства WS (объем памяти, занимаемой процессом). Наконец, третий командлет Select-Object выбирает из упорядоченного массива объекта *последние пять элементов*.

Предположим теперь, что нам нужно получить список запущенных в системе процессов, в котором были бы указаны только имена процессов и их идентификаторы. Если вы не помните названия нужных свойств, то можно с помощью командлета Get-Member вновь просмотреть структуру возвращаемых командой Get-Process объектов:

```
PS C:\> Get-Process | Get-Member -MemberType Property

   TypeName: System.Diagnostics.Process

Name                MemberType Definition
----                ---------- ----------
BasePriority        Property   System.Int32 BasePriority {get;}
. . .
Id                  Property   System.Int32 Id {get;}
. . .
ProcessName         Property   System.String ProcessName {get;}
. . .
WorkingSet64        Property   System.Int64 WorkingSet64 {get;}
```

Итак, в итоговых объектах нам нужно оставить только свойства ProcessName и Id. Это можно сделать, указав имена нужных свойств в качестве параметров командлета Select-Object:

```
PS C:\> Get-Process | Select-Object ProcessName, Id

ProcessName                 Id
-----------                 --
AcroRd32                  1752
alg                        256
ati2evxx                  1632
atiptaxx                  1988
csrss                      804
ctfmon                     872
explorer                   740
Far                       2084
Idle                         0
kav                        884
kavsvc                    1756
klswd                     1792
lsass                      892
outpost                   1784
powershell                2904
. . .
```

Посмотрим теперь, какой тип имеет объект, формируемый в конвейере командлетом Select-Object, и какие свойства имеются у этого объекта:

```
PS C:\> Get-Process | Select-Object ProcessName, Id | Get-Member

   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       System.Boolean Equals(Object obj)
GetHashCode Method       System.Int32 GetHashCode()
GetType     Method       System.Type GetType()
ToString    Method       System.String ToString()
Id          NoteProperty System.Int32 Id=1752
ProcessName NoteProperty System.String ProcessName=AcroRd32
```

Как видите, выходной объект имеет тип System.Management.Automation.PSCustomObject (напомним, что командлет Get-Process возвращал объекты типа System.Diagnostics.Process) и у него имеются только два свой-

ства ProcessName и Id. Это связано с тем, что при использовании командлета Select-Object для выбора указанных свойств он копирует значения этих свойств из объектов, поступающих по конвейеру ему на вход, и создает новые объекты, которые содержат указанные свойства со скопированными значениями.

Командлет Select-Object может не только удалять из объектов ненужные свойства, но и добавлять новые *вычисляемые* свойства. Для этого новое свойство нужно представить в виде *хэш-таблицы*, где первый элемент (ключ Name) соответствует имени добавляемого свойства, а второй элемент (ключ Expression) — значению этого свойства для текущего элемента конвейера.

> **ЗАМЕЧАНИЕ**
>
> Более подробно хэш-таблицы рассматриваются в *главе 6*.

Например, результатом выполнения следующего конвейера команд станет массив объектов, имеющих свойства ProcessName (имя запущенного процесса) и StartMin (минута, когда был запущен процесс):

```
PS C:\> Get-Process | Select-Object ProcessName, @{Name="StartMin";
Expression = {$_.StartTime.Minute}}
ProcessName           StartMin
-----------           --------
alg                         45
ati2evxx                    45
atiptaxx                    48
csrss                       45
ctfmon                      48
explorer                    48
. . .
```

Здесь свойство StartMin является вычисляемым, его значение для каждого элемента конвейера задается блоком кода {$_.StartTime.Minute}, где переменная $_ соответствует текущему объекту конвейера.

Выполнение произвольных действий над объектами в конвейере (командлет *ForEach-Object*)

Командлет ForEach-Object позволяет выполнить определенный блок сценария (код на языке PowerShell) для каждого объекта в конвейере. Другими словами, с помощью данного командлета можно производить произвольные операции над элементами конвейера.

Для примера давайте подсчитаем общий объем файлов, хранящихся в корневом каталоге диска c:. Для этого сначала перейдем в данный каталог и объявим переменную `$TotalLength` и обнулим ее:

```
PS C:\Documents and Settings\404_Popov:\> cd c:\
PS C:\> $TotalLength=0
```

Теперь выполним команду `dir` и результат ее работы передадим по конвейеру командлету `ForEach-Object`:

```
PS C:\> dir | ForEach-Object {$TotalLength+=$_.Length}
```

В блоке сценария командлета `ForEach-Object` к текущему значению переменной `$TotalLength` прибавляется значение свойства `Length` проходящего через конвейер объекта (размер соответствующего этому объекту файла). В результате в переменной `$TotalLength` будет храниться общий размер файлов в байтах:

```
PS C:\> $TotalLength
228528499
```

Группировка объектов (командлет *Group-Object*)

Проходящие по конвейеру объекты можно сгруппировать по значению определенных свойств с помощью командлета `Group-Object`. В одну группу будут попадать объекты, имеющие одинаковые значения указанных свойств (свойства могут быть вычисляемыми).

Рассмотрим пример. Командлет `Get-Process` генерирует объекты, имеющие свойства `Company` (название компании-разработчика определенного модуля, запущенного в операционной системе в качестве процесса). Выполним группировку этих объектов по значению свойства `Company`:

```
PS C:\> Get-Process | Group-Object Company

Count Name                      Group
----- ----                      -----
    1 Adobe Systems Incorpor... {AcroRd32}
   13 Microsoft Corporation     {alg, ctfmon, lsass, OUTLOOK,...}
    7                           {csrss, Idle, kav, kavsvc,...}
    5 Корпорация Майкрософт     {explorer, MAPISP32, scardsvr,...}
    1 Eugene Roshal & FAR Group {Far}
    2 Intel Corporation         {hkcmd, igfxpers}
    1 Корпорация Microsoft      {jview}
```

```
    1 Kaspersky Lab             {klnagent}
    1 Visioneer Inc             {OneTouchMon}
    1 Hewlett-Packard           {sdlaunch}
    1 Realtek Semiconductor ... {SOUNDMAN}
```

Как видите, в колонке `Count` отображается количество элементов в каждой из групп, а в колонке `Group` перечислены элементы, входящие в группы.

Если нужно просто узнать количество элементов в группах, можно запустить командлет `Group-Object` с параметром `-NoElement`:

```
PS C:\> Get-Process | Group-Object Company -NoElement

Count Name
----- ----
    1 Adobe Systems Incorpor...
   13 Microsoft Corporation
    7
    5 Корпорация Майкрософт
    1 Eugene Roshal & FAR Group
    2 Intel Corporation
    1 Корпорация Microsoft
    1 Kaspersky Lab
    1
    1 Visioneer Inc
    1 Hewlett-Packard
    1 Realtek Semiconductor ...
```

Измерение характеристик объектов (командлет *Measure-Object*)

В PowerShell имеется еще один полезный командлет `Measure-Object`, предназначенный для выполнения функций агрегирования (сумма, выбор минимального, максимального или среднего значения) над свойствами элементов в конвейере объектов.

Рассмотрим пример. Ранее мы уже находили общий размер файлов в корневом каталоге диска C:, применяя для этого командлет `ForEach-Object`:

```
PS C:\> $TotalLength=0
PS C:\> dir | ForEach-Object {$TotalLength+=$_.length}
PS C:\> $TotalLength
228528499
```

С помощью командлета `Measure-Object` мы также сможем найти суммарный размер файлов. Для этого нужно указать, что `Measure-Object` должен для всех элементов конвейера просуммировать (параметр `-Sum`) значения свойства `Length`:

```
PS C:\> dir | Measure-Object -Property Length -Sum

Count    : 44
Average  :
Sum      : 228528499
Maximum  :
Minimum  :
Property : Length
```

Результат будет выведен в поле `Sum`. Для выполнения других операций нужно указать соответствующий параметр: `-Average` для нахождения среднего значения, `-Minimum` или `-Maximum` для нахождения минимального или максимального значения соответственно:

```
PS C:\> dir | Measure-Object -Property Length -Minimum -Maximum
-Average -Sum

Count    : 44
Average  : 5201773.63636364
Sum      : 228528499
Maximum  : 226324357
Minimum  : 0
Property : length
```

Также с помощью командлета `Measure-Object` можно получать статистическую информацию о текстовых файлах: количество строк, слов и символов.

Вызов статических методов

Иногда при работе в PowerShell возникает необходимость воспользоваться методами, которые определены в классах (типах) .NET, не создавая и не используя экземпляры этих классов. Классы, содержащие только такие методы, называются *статическими*, так как они не создаются, не уничтожаются и не меняются. В частности, статическим является класс `System.Math`, методы которого часто используются для математических вычислений.

Для обращения к статическому классу его имя следует заключить в квадратные скобки, например:

```
PS C:\> [System.Math]

IsPublic IsSerial Name       BaseType
-------- -------- ----       --------
True     False    Math       System.Object
```

Методы статического класса также называются статическими. Для просмотра доступных статических методов класса нужно передать имя этого класса (в квадратных скобках) по конвейеру командлету `Get-Member` с параметром `-Static`:

```
PS C:\> [System.Math] | Get-Member -Static

   TypeName: System.Math

Name             MemberType Definition
----             ---------- ----------
Abs              Method     static System.Single Abs(Single va. . .
Acos             Method     static System.Double Acos(Double d
Asin             Method     static System.Double Asin(Double d
Atan             Method     static System.Double Atan(Double d
Atan2            Method     static System.Double Atan2(Double . . .
BigMul           Method     static System.Int64 BigMul(Int32 a. . .
Ceiling          Method     static System.Double Ceiling(Doubl. . .
Cos              Method     static System.Double Cos(Double d)
Cosh             Method     static System.Double Cosh(Double v
DivRem           Method     static System.Int32 DivRem(Int32 a . .
Equals           Method     static System.Boolean Equals(Objec . .
Exp              Method     static System.Double Exp(Double d)
Floor            Method     static System.Double Floor(Double . . .
IEEERemainder    Method     static System.Double IEEERemainder. . .
Log              Method     static System.Double Log(Double d). . .
Log10            Method     static System.Double Log10(Double
Max              Method     static System.SByte Max(SByte val1. . .
Min              Method     static System.SByte Min(SByte val1. . .
Pow              Method     static System.Double Pow(Double x,. . .
ReferenceEquals  Method     static System.Boolean ReferenceEqu. . .
```

```
Round           Method      static System.Double Round(Double . . .
Sign            Method      static System.Int32 Sign(SByte val. . .
Sin             Method      static System.Double Sin(Double a)
Sinh            Method      static System.Double Sinh(Double v
Sqrt            Method      static System.Double Sqrt(Double d
Tan             Method      static System.Double Tan(Double a)
Tanh            Method      static System.Double Tanh(Double v
Truncate        Method      static System.Decimal Truncate(Dec. . .
E               Property    static System.Double E {get;}
PI              Property    static System.Double PI {get;}
```

Как видите, методы класса `System.Math` реализуют различные математические функции, их легко распознать по названию.

Для доступа к определенному статическому методу или свойству используются два идущих подряд двоеточия (::), а не точка (.), как в обычных объектах. Например, для вычисления квадратного корня из числа (статического метода `Sqrt`) и сохранения результата в переменную используется следующая конструкция:

```
PS C:\> $a=[System.Math]::Sqrt(25)
PS C:\> $a
5
```

Управление выводом команд в PowerShell

Ранее мы уже упоминали, что в PowerShell имеется база данных (набор XML-файлов), содержащая *модули форматирования* по умолчанию для различных типов .NET-объектов. Эти модули определяют, какие свойства объекта отображаются при выводе и в каком формате: списка или таблицы. (Напомним, что командлеты PowerShell возвращают .NET-объекты, которые, как правило, не знают, каким образом отображать себя на экране.) Когда объект достигает конца конвейера, PowerShell определяет его тип и ищет его в списке объектов, для которых определено правило форматирования. Если данный тип в списке обнаружен, то к объекту применяется соответствующий модуль форматирования; если нет, то PowerShell просто отображает свойства этого .NET-объекта.

Также в PowerShell можно явно задавать правила форматирования данных, выводимых командлетами, и, как и в командном интерпретаторе cmd.exe, перенаправлять эти данные в файл, на принтер или в пустое устройство.

Форматирование выводимой информации

В традиционных оболочках команды и утилиты сами форматируют выводимые данные. Некоторые команды (например, `dir` в интерпретаторе cmd.exe) позволяют настраивать формат вывода с помощью специальных параметров.

В оболочке PowerShell вывод форматируют только четыре специальных командлета `Format` (табл. 5.3). Это упрощает изучение, так как не нужно запоминать средства и параметры форматирования для других команд (остальные командлеты вывод не форматируют).

Таблица 5.3. *Командлеты PowerShell для форматирования вывода*

Командлет	Описание
`Format-Table`	Форматирует вывод в виде таблицы, столбцы которой содержат свойства объекта (также могут быть добавлены вычисляемые столбцы). Поддерживается возможность группировки выводимых данных
`Format-List`	Выводит объект как список свойств. При этом каждое свойство отображается на новой строке. Поддерживается возможность группировки выводимых данных
`Format-Custom`	Использует пользовательское представление (view) для форматирования вывода
`Format-Wide`	Форматирует объекты в виде широкой таблицы, в которой отображается только одно свойство каждого объекта

Если ни один из командлетов `Format` явно не указан, то используется модуль форматирования по умолчанию, который определяется по типу отображаемых данных. Например, при выполнении командлета `Get-Service` данные по умолчанию выводятся как таблица с тремя столбцами (`Status`, `Name` и `DisplayName`):

```
PS C:\> Get-Service

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Оповещатель
Running  ALG                Служба шлюза уровня приложения
Stopped  AppMgmt            Управление приложениями
Stopped  aspnet_state       ASP.NET State Service
Running  Ati HotKey Poller  Ati HotKey Poller
Running  AudioSrv           Windows Audio
```

```
Running  BITS              Фоновая интеллектуальная служба пер...
Running  Browser           Обозреватель компьютеров
Stopped  cisvc             Служба индексирования
Stopped  ClipSrv           Сервер папки обмена
Stopped  clr_optimizatio... .NET Runtime Optimization Service v...
Stopped  COMSysApp         Системное приложение COM+
Running  CryptSvc          Службы криптографии
Running  DcomLaunch        Запуск серверных процессов DCOM
Running  Dhcp              DHCP-клиент
. . .
```

Для изменения формата выводимых данных нужно направить их по конвейеру соответствующему командлету `Format`. Например, следующая команда выведет список служб с помощью командлета `Format-List`:

```
PS C:\> Get-Service | Format-List

Name                  : Alerter
DisplayName           : Оповещатель
Status                : Stopped
DependentServices     : {}
ServicesDependedOn    : {LanmanWorkstation}
CanPauseAndContinue   : False
CanShutdown           : False
CanStop               : False
ServiceType           : Win32ShareProcess

Name                  : ALG
DisplayName           : Служба шлюза уровня приложения
Status                : Running
DependentServices     : {}
ServicesDependedOn    : {}
CanPauseAndContinue   : False
CanShutdown           : False
CanStop               : True
ServiceType           : Win32OwnProcess
. . .
```

Как видите, при использовании формата списка выводится больше сведений о каждой службе, чем в формате таблицы (вместо трех столбцов данных о каждой службе в формате списка выводятся девять строк данных). Однако это вовсе не означает, что командлет `Format-List` извлекает дополнитель-

ные сведения о службах. Эти данные содержатся в объектах, возвращаемых командой `Get-Service`, однако используемый по умолчанию командлет `Format-Table` отбрасывает их, потому что не может вывести на экран более трех столбцов.

При форматировании вывода с помощью командлетов `Format-List` и `Format-Table` можно указывать имена свойств, которые должны быть отображены (напомним, что просмотреть список свойств, имеющихся у объекта, позволяет командлет `Get-Member`). Например:

```
PS C:\> Get-Service | Format-List Name, Status, CanStop

Name    : Alerter
Status  : Stopped
CanStop : False

Name    : ALG
Status  : Running
CanStop : True

Name    : AppMgmt
Status  : Stopped
CanStop : False
. . .
```

Вывести все имеющиеся у объектов свойства можно с помощью параметра *, например:

```
PS C:\> Get-Service | Format-List *

Name                  : Alerter
CanPauseAndContinue   : False
CanShutdown           : False
CanStop               : True
DisplayName           : Оповещатель
DependentServices     : {}
MachineName           : .
ServiceName           : Alerter
ServicesDependedOn    : {LanmanWorkstation}
ServiceHandle         : SafeServiceHandle
Status                : Running
ServiceType           : Win32ShareProcess
Site                  :
Container             :
```

```
Name                  : ALG
CanPauseAndContinue   : False
CanShutdown           : False
. . .
```

Перенаправление выводимой информации

В оболочке PowerShell имеются несколько командлетов, с помощью которых можно управлять выводом данных. Эти командлеты начинаются со слова `Out`, их список можно увидеть следующим образом:

```
PS C:\> Get-Command out-* | Format-Table Name

Name
----
Out-Default
Out-File
Out-Host
Out-Null
Out-Printer
Out-String
```

По умолчанию выводимая информация передается командлету `Out-Default`, который, в свою очередь, делегирует всю работу по выводу строк на экран командлету `Out-Host`. Для понимания данного механизма нужно учитывать, что архитектура PowerShell подразумевает различие между собственно ядром оболочки (интерпретатором команд) и главным приложением (host), которое использует это ядро. В принципе, в качестве главного может выступать любое приложение, в котором реализован ряд специальных интерфейсов, позволяющих корректно интерпретировать получаемую от PowerShell информацию. *В нашем случае главным приложением является консольное окно*, в котором мы работаем с оболочкой, и командлет `Out-Host` передает выводимую информацию в это консольное окно.

Параметр `-Paging` командлета `Out-Host`, подобно команде `more` интерпретатора cmd.exe, позволяет организовать постраничный вывод информации, например:

```
PS C:\> Get-Help Get-Process -Full | Out-Host -Paging
ИМЯ
    Get-Process

ОПИСАНИЕ
    Отображает процессы, выполняющиеся на локальном компьютере.
```

```
СИНТАКСИС
    Get-Process [[-name] <string[]>] [<CommonParameters>]

. . .
<ПРОБЕЛ> следующая страница; <CR> следующая строка; Q выход
```

Сохранение данных в файл

Как уже упоминалось ранее, PowerShell поддерживает перенаправление вывода команд в текстовые файлы с помощью стандартных операторов > и >>. Например, следующая команда выведет содержимое корневого каталога c:\ в текстовый файл d:\dir_c.txt (если такой файл существовал, то он будет перезаписан):

```
PS C:\> dir c:\ > d:\dir_c.txt
```

Если нужно перенаправить вывод команды в файл в режиме добавления (с сохранением прежнего содержимого данного файла), следует воспользоваться оператором >>:

```
PS C:\> dir c:\ >> d:\dir_c.txt
```

Кроме операторов перенаправления > и >> в PowerShell имеется командлет `Out-File`, также позволяющий направить выводимые данные вместо окна консоли в текстовый файл. При этом командлет `Out-File` имеет несколько дополнительных параметров, с помощью которых можно более гибко управлять выводом: задавать тип кодировки файла, задавать длину выводимых строк в знаках, выбирать режим перезаписи файла (табл. 5.4).

Таблица 5.4. *Некоторые параметры командлета* `Out-File`

Параметр	Описание
`-FilePath`	Указывает путь к выходному файлу
`-Encoding`	Определяет кодировку выходного файла. Допустимые значения: `Unicode`, `UTF7`, `UTF8`, `UTF32`, `ASCII`, `BigEndianUnicode`, `Default` и `OEM`. По умолчанию в PowerShell используется кодировка `Unicode`. Для сохранения текста в Windows-кодировке следует выбирать значение `Default` (кодировка текущей кодовой страницы ANSI), для сохранения текста в DOS-кодировке — значение `OEM`
`-Width`	Указывает число знаков в каждой выходной строке
`-Append`	Записывает выходные данные в конец существующего файла, а не замещает его содержимое
`-NoClobber`	Задает режим перезаписи файла. При указании этого параметра, если выходной файл уже существует, он не будет перезаписываться (по умолчанию, если файл существует по указанному пути, командлет `Out-File` перезаписывает его без предупреждения). Если одновременно используются параметры `-Append` и `-NoClobber`, выходные данные записываются в конец существующего файла

Например, следующая команда сохранит в файле c:\help.txt детальный вариант встроенной справки по командлету Get-Process (файл c:\help.txt будет создан в Windows-кодировке):

```
PS C:\> Get-Help Get-Process -Detailed | Out-File -FilePath c:\help.txt
-Encoding "Default"
```

Печать данных

Данные можно вывести непосредственно на принтер с помощью командлета Out-Printer. При этом печать может производиться как на принтере по умолчанию (никаких специальных параметров для этого указывать не нужно), так и на произвольном принтере (в этом случае отображаемое имя принтера должно быть указано в качестве значения параметра -Name). Например:

```
PS C:\script> Get-Process | Out-Printer -Name "Xerox Phaser 3500 PCL 6"
```

Подавление вывода

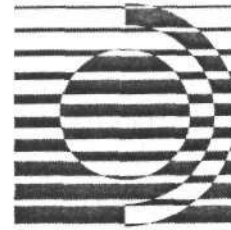
Командлет Out-Null служит для отбрасывания любых своих входных данных. Это может пригодиться для подавления вывода на экран ненужных сведений, полученных в качестве побочного эффекта выполнения какой-либо команды. Например, при создании каталога командой mkdir на экран выводится его содержимое:

```
PS C:\> mkdir klop


    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         03.12.2007      1:01            klop
```

Если вы не желаете видеть эту информацию, то результат выполнения команды mkdir нужно передать по конвейеру командлету Out-Null:

```
PS C:\> mkdir klop | Out-Null
PS C:\>
```

Как видите, в этом случае никаких сообщений на экран не выводится.



Глава 6

Переменные, массивы и хэш-таблицы

Во многих приведенных ранее примерах мы уже использовали различные числовые и символьные литералы (константы), а также переменные PowerShell, сохраняя в них результаты выполнения команд. Кроме переменных в PowerShell, как во многих других языках программирования, поддерживаются массивы, а также более специфические структуры — ассоциативные массивы (хэш-таблицы).

Рассмотрим эти элементы языка PowerShell более подробно.

Числовые и символьные литералы

Практически в каждом языке программирования имеется возможность работы с числами и символьными строками, причем способы их задания могут быть различными (например, в Basic строки нужно заключать в двойные кавычки, а в Delphi — в одинарные). PowerShell также поддерживает целые и вещественные числа, а также символьные строки нескольких видов.

Числовые литералы

Язык PowerShell поддерживает все основные числовые типы платформы .NET: `System.Int32`, `System.Int64`, `System.Double`. При этом явно задавать тип чисел нет необходимости — система сама выбирает подходящий тип для указываемого вами числа. Проверим тип нескольких чисел, воспользовавшись для этого методом `GetType`:

```
PS C:\> (10).GetType().FullName
System.Int32
PS C:\> (10.23).GetType().FullName
System.Double
PS C:\> (10+10.23).GetType().FullName
System.Double
```

В PowerShell предусмотрены специальные *суффиксы-множители* для упрощения работы с величинами, часто используемыми системными администраторами: килобайтами, мегабайтами и гигабайтами (табл. 6.1).

Таблица 6.1. *Суффиксы-множители в PowerShell*

Суффикс-множитель	Числовой множитель	Пример	Числовое значение для примера
KB	1024	2KB	2048
kb	1024	1.1kb	1126.4
MB	1024*1024	3MB	3 145 728
mb	1024*1024	2.5mb	2 621 440
GB	1024*1024*1024	1GB	1 073 741 824
gb	1024*1024*1024	2.23gb	2 394 444 267.52

Приведем примеры:

```
PS C:\> 1mb+10kb
1058816
PS C:\> 2GB+56MB
2206203904
```

В PowerShell можно оперировать числами в шестнадцатеричном формате, используя для этого те же обозначения, что и в С-подобных языках программирования: перед числом указывается префикс `0x`, а в записи числа могут присутствовать цифры и буквы `A`, `B`, `C`, `D`, `E` и `F` (независимо от регистра). Например:

```
PS C:\> 0x10
16
PS C:\> 0xA
10
PS C:\> 0xcd
205
```

Символьные строки

Все символьные строки в PowerShell являются объектами типа `System.String` и представляют собой последовательность 32-битовых символов в кодировке Unicode. Длина строк не ограничена, содержимое строк нельзя изменять (можно только копировать).

В PowerShell поддерживается четыре вида символьных строк.

Строки в одинарных и двойных кавычках

Строки могут задаваться последовательностью символов, заключенных в одинарные или двойные кавычки:

```
PS C:\> 'Строка в одинарных кавычках'
Строка в одинарных кавычках
PS C:\> "Строка в двойных кавычках"
Строка в двойных кавычках
```

Строки могут содержать любые символы (в том числе символы разрыва строки и возврата каретки), кроме соответствующего одиночного закрывающего символа (одинарной или двойной кавычки). Строка в одинарных кавычках может содержать двойные кавычки и наоборот:

```
PS C:\> 'Строка в "одинарных" кавычках'
Строка в "одинарных" кавычках
PS C:\> "Строка в 'двойных' кавычках"
Строка в 'двойных' кавычках
```

Если внутри строки нужно поместить символ, ограничивающий данную строку (то есть одинарную или двойную кавычку), то нужно написать этот символ два раза подряд:

```
PS C:\> 'Строка в ''одинарных кавычках'
Строка в 'одинарных кавычках
PS C:\> "Строка в ""двойных кавычках"
Строка в "двойных кавычках
```

Строки в двойных кавычках являются *расширяемыми*. Это означает, что если внутри строки в двойных кавычках встречается имя переменной или другое выражение, которое может быть вычислено, то в данную строку подставляется значение данной переменной или результат вычисления выражения. Например:

```
PS C:\> $a=123
PS C:\> "$a равно $a"
123 равен 123
```

Если имя переменной встречается внутри строки в одинарных кавычках, то никакой подстановки значения переменной не происходит:

```
PS C:\> '$a равно $a'
$a равен $a
```

При необходимости можно отключить расширение определенной переменной внутри строки в двойных кавычках. Для этого перед знаком $ этой переменной нужно указать символ обратного апострофа `, например:

```
PS C:\> "`$a равно $a"
$a равно 123
```

Символы, имеющие специальное значение, вставляются в строки в двойных кавычках с помощью escape-последовательностей, которые в PowerShell начинаются с символа обратного апострофа ` (табл. 6.2).

Таблица 6.2. *Escape-последовательности PowerShell*

Переменная	Описание
`n	Разрыв строки
`r	Возврат каретки
`t	Горизонтальная табуляция
`a	Звуковой сигнал
`b	Забой (backspace)
`'	Одинарная кавычка
`"	Двойная кавычка
`0	Пустой символ (null)
``	Обратный апостроф

Замечание

В других языках программирования типа C, C#, JScript или Perl для выделения специальных символов (escape-последовательностей) используется обратная косая черта (например, \n или \t). Разработчики оболочки PowerShell приняли решение ввести другой символ для escape-последовательностей, чтобы избежать проблем при использовании символа \ в качестве разделителя компонентов пути в файловой системе Windows и других пространствах имен PowerShell.

Вставим символ разрыва строки в строку в двойных кавычках:

```
PS C:\> "Строка в `nдвойных кавычках"
Строка в
двойных кавычках
```

Как видите, на экран информация выводится в двух строках. Если же вставить escape-последовательность `n в строку в одинарных кавычках, то разрыва строки не произойдет:

```
PS C:\> 'Строка в `nодинарных кавычках'
Строка в `nодинарных кавычках
```

Кроме переменных в расширяемых строках могут указываться так называемые *подвыражения* (subexpression) — ограниченные символами $(...) фрагменты кода на языке PowerShell, которые в строках заменяются на результаты вычисления этих фрагментов. Например:

```
PS C:\> "3+2 равно $(3+2)"
3+2 равно 5
```

Автономные строки

В PowerShell наряду с обычными строками в одинарных и двойных кавычках поддерживаются так называемые *автономные строки* (также известные под именем строк типа "here-string"). Подобные строки обычно используются для вставки в сценарий больших блоков текста или при генерации текстовой информации для других программ и имеют следующий формат:

```
@<кавычка><разрыв_строки>блок текста<разрыв_строки><кавычка>@
```

Кавычки могут быть как одинарными, так и двойными, при этом смысл их остается тем же, что и для обычных строк: переменные и подвыражения, стоящие внутри двойных кавычек, заменяются их значениями, а стоящие внутри одинарных кавычек остаются неизменными. Например:

```
PS C:\> $a=@"
>> 1 Первая строка
>> $(1+1) Вторая строка
>> "Третья строка"
>> "@
>>
PS C:\> $a
1 Первая строка
2 Вторая строка
"Третья строка"
PS C:\> $a=@'
>> 1 Первая строка
>> $(1+1) Вторая строка
>> 'Третья строка'
>> '@
>>
PS C:\> $a
1 Первая строка
$(1+1) Вторая строка
'Третья строка'
```

Обратите внимание, что ограничитель автономных строк обязательно должен содержать символ разрыва строки, поэтому внутри таких строк различные специальные символы (например, одинарные или двойные кавычки) могут применяться без ограничений.

Переменные PowerShell

Как мы уже знаем, имена переменных PowerShell всегда начинаются со знака доллара ($). Переменные PowerShell не нужно предварительно объявлять или описывать, они создаются при первом присваивании переменной значения.

Если попытаться обратиться к несуществующей переменной, то система вернет значение `$Null`.

> ***Замечание***
>
> `$Null`, как `$True` и `$False`, является специальной переменной, определенной в системе. Изменить значения этих переменных нельзя.

Проверить, определена ли переменная, можно с помощью командлета `Test-Path`. Например, следующая команда проверяет, существует ли переменная `MyVariable`:

```
PS C:\> Test-Path Variable:MyVariable
False
```

Список всех переменных, определенных в текущем сеансе работы, можно увидеть, обратившись к виртуальному диску PowerShell `Variable:` с помощью команды `dir`:

```
PS C:\> dir Variable:

Name                           Value
----                           -----
Error                          {DriveNotFound,Microsoft.Power. . .
DebugPreference                SilentlyContinue
PROFILE                        C:\Documents and Settings\404_. . .
HOME                           C:\Documents and Settings\404_. . .
Host                           System.Management.Automation.I. . .
MaximumHistoryCount            64
MaximumAliasCount              4096
input                          System.Array+SZArrayEnumerator
ReportErrorShowSource          1
. . .
MaximumDriveCount              4096
MaximumVariableCount           4096
$                              variable:MyVariable
```

Если пользователь не создавал пока своих переменных, то в системе определены только *переменные оболочки PowerShell*.

Переменные оболочки PowerShell

Переменные оболочки — это набор переменных, которые создаются, объявляются оболочкой PowerShell и присутствуют по умолчанию в каждом сеансе работы. Переменные оболочки сохраняются в течение всего сеанса и доступны всем командам, сценариям и приложениям, которые выполняются в данном сеансе.

Поддерживаются два вида переменных оболочки.

- **Автоматические переменные.** В этих переменных хранятся параметры *состояния оболочки* PowerShell. Автоматические переменные сохраняются и динамически изменяются самой системой. Пользователи не могут (и не должны) изменять значения этих переменных.

  Например, значением переменной `$PID` является идентификатор текущего процесса PowerShell.exe.

- **Переменные настроек.** В этих переменных хранятся настройки активного пользователя. Эти переменные создаются оболочкой PowerShell и заполняются значениями по умолчанию. Пользователи могут изменять значения этих переменных.

  Например, переменная `$MaximumHistoryCount` определяет максимальное число записей в журнале сеанса.

В табл. 6.3 приведено краткое описание переменных оболочки.

Таблица 6.3. *Переменные оболочки PowerShell*

Переменная	Описание
`$$`	Содержит последнюю лексему последней полученной оболочкой строки
`$?`	Показывает, успешно ли завершилась последняя операция
`$^`	Содержит первую лексему последней полученной оболочкой строки
`$_`	При использовании в блоках сценариев, фильтрах и инструкции `Where` содержит текущий объект конвейера
`$Args`	Содержит массив параметров, передаваемых в функцию
`$DebugPreference`	Указывает действие, которое необходимо выполнить при записи данных с помощью командлета `Write-Debug`

Таблица 6.3 *(продолжение)*

Переменная	Описание
`$Error`	Содержит объекты, для которых возникла ошибка при обработке в командлете
`$ErrorActionPreference`	Указывает действие, которое необходимо выполнить при записи данных с помощью командлета `Write-Error`
`$ForEach`	Обращается к итератору в цикле `ForEach`
`$Home`	Указывает домашний каталог пользователя. Эквивалент конструкции %HomeDrive%%HomePath% в оболочке cmd.exe
`$Input`	Используется в блоках сценариев, находящихся в конвейере
`$LASTEXITCODE`	Содержит код завершения последнего выполнения исполнимого файла Win32
`$MaximumAliasCount`	Хранит максимальное число псевдонимов, доступных сеансу
`$MaximumDriveCount`	Содержит максимальное число доступных дисков, за исключением предоставляемых операционной системой
`$MaximumFunctionCount`	Показывает максимальное число функций, доступных сеансу
`$MaximumHistoryCount`	Указывает максимальное число записей, сохраненных в истории команд
`$MaximumVariableCount`	Содержит максимальное число переменных, доступных сеансу
`$PSHome`	Показывает имя каталога, в который установлен PowerShell
`$Host`	Содержит сведения о текущем узле
`$OFS`	Используется в качестве разделителя при преобразовании массива в строку. По умолчанию данная переменная имеет значение пробела
`$ReportErrorShowExceptionClass`	Если значение переменной равно `$True`, показывает имена класса выведенных исключений

Таблица 6.3 (окончание)

Переменная	Описание
`$ReportErrorShowInnerException`	Если значение переменной равно `$True`, показывает цепочку внутренних исключений. Вывод каждого исключения управляется теми же параметрами, что и корневого исключения, то есть параметры `$ReportErrorShow*` используются для вывода каждого исключения
`$ReportErrorShowSource`	Если значение переменной равно `$True`, показывает имена сборок выведенных исключений
`$ReportErrorShowStackTrace`	Если значение переменной равно `$True`, происходит трассировка стека исключений
`$StackTrace`	Содержит подробные сведения трассировки стека на момент последней ошибки
`$VerbosePreference`	Указывает действие, которое нужно выполнить, если данные записываются с помощью командлета `Write-Verbose`
`$WarningPreference`	Указывает действие, которое необходимо выполнить при записи данных с помощью командлета `Write-Warning` в сценарии

Переменными оболочки можно пользоваться так же, как и другими видами переменных. Например, следующая команда выведет на экран содержимое домашнего каталога PowerShell, путь к которому хранится в переменной оболочки `$PSHome`:

```
PS C:\> dir $PSHome

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\system32\
WindowsPowerShell\v1.0

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----          01.12.2006     19:36            examples
d----          21.05.2007     16:00            ru
-----          08.09.2006     12:28      22120 certificate.form. . .
-----          08.09.2006     12:28      60703 dotnettypes.form. . .
-----          08.09.2006     12:28      19730 filesystem.forma. . .
-----          08.09.2006     12:28     250197 help.format.ps1xml
```

```
-----        13.10.2006     13:24      330240 powershell.exe
-----        08.09.2006     12:28       65283 powershellcore.for. . .
-----        08.09.2006      2:28       13394 powershelltrace.fo. . .
-----        09.10.2006     13:26        4608 pwrshmsg.dll
-----        13.10.2006     18:04       20992 pwrshsip.dll
-----        08.09.2006     12:28       13540 registry.format.ps1xml
-----        08.09.2006     12:28      129836 types.ps1xml
```

Пользовательские переменные. Типы переменных

Пользовательская переменная создается после первого присваивания ей значения. Например, создадим целочисленную переменную `$a`:

```
PS C:\> $a=1
PS C:\> $a
1
PS C:\> Test-Path Variable:a
True
PS C:\> dir Variable:a

Name                           Value
----                           -----
a                              1
```

Проверим, какой тип имеет переменная `$a`. Для этого можно воспользоваться командлетом `Get-Member` или методом `GetType()`:

```
PS C:\> $a | Get-Member
   TypeName: System.Int32
. . .
PS C:\> $a.GetType().FullName
System.Int32
```

Итак, переменная `$a` сейчас имеет тип `System.Int32`. Присвоим этой переменной другое значение (строку) и вновь проверим тип:

```
PS C:\> $a="aaa"
PS C:\> $a | Get-Member
   TypeName: System.String
. . .
```

Как видите, тип переменной `$a` изменился на `System.String`, то есть тип переменной определяется типом последнего присвоенного ей значения.

Можно также явно указать тип переменной при ее определении, указав в квадратных скобках соответствующий атрибут типа. При этом выражение, стоящее в правой части после знака равенства, будет преобразовано (если это возможно) к данному типу. Например, объявим целочисленную переменную `$a` и присвоим этой переменной символьное значение, которое можно преобразовать к целому типу:

```
PS C:\> [System.Int32]$a=10
PS C:\> $a="123"
PS C:\> $a
123
PS C:\> $a.GetType().FullName
System.Int32
```

Как видите, строка "123" была преобразована в целое число 123. Если же попытаться записать в переменную `$a` значение, которое не может быть преобразовано в целое число, то возникнет ошибка:

```
PS C:\> $a="aaa"
Не удается преобразовать значение "aaa" в тип "System.Int32".
Ошибка: "Input string was not in a correct format."
В строка:1 знак:3
+ $a= <<<< "aaa"
```

Вместо явного указания .NET-типа переменной можно пользоваться более краткими *псевдонимами типов*. Например:

```
PS C:\> [int]$a=10
PS C:\> $a.GetType().FullName
System.Int32
```

Наиболее часто используемые псевдонимы типов приведены в табл. 6.4.

Таблица 6.4. *Псевдонимы типов PowerShell*

Псевдоним типа	Соответствующий .NET-тип
`[int]`	`System.Int32`
`[int[]]`	`System.Int32[]` (массив элементов типа `System.Int32`)
`[long]`	`System.Int64`
`[long[]]`	`System.Int64[]` (массив элементов типа `System.Int64`)
`[string]`	`System.String`
`[string[]]`	`System.String[]` (массив элементов типа `System.String`)

Таблица 6.4 (окончание)

Псевдоним типа	Соответствующий .NET-тип
[char]	System.Char
[char[]]	System.Char[] (массив элементов типа System.Char)
[bool]	System.Boolean
[bool[]]	System.Boolean[] (массив элементов типа System.Boolean)
[byte]	System.Byte
[byte[]]	System.Byte[] (массив элементов типа System.Byte)
[double]	System.Double
[double[]]	System.Double[] (массив элементов типа System.Double)
[decimal]	System.Decimal
[decimal[]]	System.Decimal[] (массив элементов типа System.Decimal)
[float]	System.Float
[single]	System.Single
[regex]	System.Text.RegularExpressions.regex
[array]	System.Array
[xml]	System.Xml.XmlDocument
[scriptblock]	System.Management.Automation.ScriptBlock
[switch]	System.Management.Automation.SwitchPazameter
[hashtable]	System.Collections.Hashtable
[psobject]	System.Management.Automation.PSObject
[type]	System.Type

Обратим внимание на псевдоним [psobject], который соответствует типу System.Management.Automation.PSObject. Объект PSObject является основой использующейся в PowerShell системы адаптации типов.

Система адаптации типов

Как уже неоднократно упоминалось ранее, ядро системы типов PowerShell составляет модель типов .NET. Однако пользователям Windows (в особенности системным администраторам) приходится иметь дело и с другими объ-

ектными моделями, поддерживаемыми операционной системой: COM, WMI, ADSI, ADO и т. д.

На самом деле команды PowerShell никогда не обращаются к объектам напрямую, а используют специальную *систему адаптации типов*, которая скрывает детали представления конкретной объектной модели. Для этого используется специальный объект `PSObject`, являющийся промежуточным уровнем между тем или иным объектом операционной системы и командой или сценарием PowerShell. Данный подход позволяет одинаково обращаться к объектам различных типов, не тратя время на изучение особенностей работы с определенной объектной моделью.

Для каждого вида данных, поддерживаемых PowerShell, в системе имеется соответствующий *адаптер типов*. Экземпляр объекта определенного типа инкапсулируется в экземпляр подходящего адаптера типов. Данный экземпляр адаптера типов служит посредником между самим объектом и системой PowerShell (рис. 6.1).

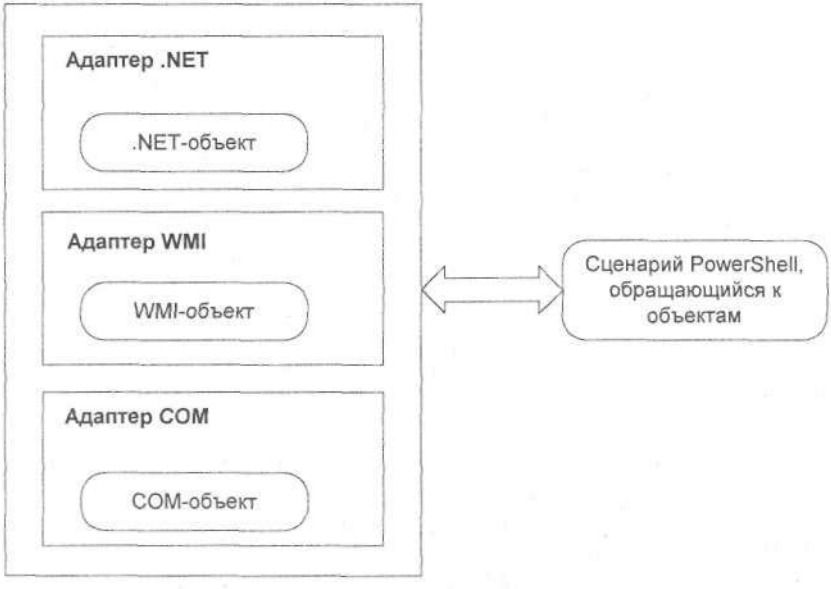


Рис. 6.1. Схема взаимодействия PowerShell с объектами различных типов

Когда вы записываете выражение типа `$x.Count`, вам не нужно знать, какой тип имеет объект, на который ссылается переменная `$x`. Нужно только, чтобы этот объект имел свойство `Count`. Система PowerShell никогда не генерирует код для прямого доступа к свойству `Count` объекта определенного типа.

Вместо этого происходит неявное обращение через объект `PSObject`, который указывает, как именно будет происходить доступ к свойству `Count`. Если в переменной `$x` хранится .NET-объект, то будет возвращено значение свойства `Length`. А если переменная `$x` содержит документ в формате XML, то адаптер XML будет искать узел с именем "Count" на верхнем уровне данного XML-документа. Объект, хранящийся в переменной `$x`, вообще может не иметь свойства `Count`, так как оболочка PowerShell позволяет определять так называемые синтетические свойства (тип `PSMember`), которые определены не в объекте, а в самой системе типов PowerShell. Список основных адаптеров типов приведен в табл. 6.5.

Таблица 6.5. *Адаптеры типов, поддерживаемые в PowerShell*

Адаптер типов	Описание
Адаптер .NET	Основной адаптер для всех .NET-типов. Данный адаптер напрямую отображает свойства соответствующего .NET-объекта и добавляет несколько новых свойств, название которых начинается с префикса PS
Адаптер COM	Обеспечивает доступ к COM-объектам, зарегистрированным в операционной системе, в том числе к объектам Windows Script Host и приложениям-серверам автоматизации типа Microsoft Word и Microsoft Excel
Адаптер WMI	Работает с объектами, возвращаемыми провайдером WMI
Адаптер ADO	Позволяет манипулировать столбцами в ADO-таблицах так, как если бы они являлись свойствами объекта
Адаптер пользовательских объектов	Управляет объектами, содержащими только синтетические свойства (таким объектам не соответствует ни один базовый тип)
Адаптер ADSI	Работает с объектами, возвращаемыми службой каталогов ADSI

Переменные среды Windows

Кроме собственных переменных, оболочка PowerShell позволяет работать и с *переменными среды* (или *переменными окружения*) Windows, каждая из которых хранится в оперативной памяти в течение всего сеанса работы операционной системы, имеет свое уникальное имя, а ее значением является строка. Стандартные переменные среды автоматически инициализируются в процессе загрузки операционной системы. К таким переменным относятся, например, `WINDIR`, которая определяет расположение каталога Windows,

TEMP, которая определяет путь к каталогу для хранения временных файлов Windows или PATH, в которой хранится *системный путь* (путь поиска), то есть список каталогов, в которых система должна искать выполняемые файлы или файлы совместного доступа (например, динамические библиотеки).

Напомним, что в оболочке cmd.exe работать с переменными среды можно с помощью команды set. Данная команда, выполненная без дополнительных параметров, возвращает список всех переменных среды:

```
C:\User> set
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\User\Application Data
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=POPOV
ComSpec=C:\WINDOWS.1\system32\cmd.exe
. . .
```

В оболочке PowerShell доступ к переменным среды можно получить через виртуальный диск Env:. Например, список всех переменных среды выводится командлетом dir:

```
PS C:\> dir env:

Name                           Value
----                           -----
Path                           C:\texmf\miktex\bin;C:\WINDOWS.1\syste3. . .
TEMP                           C:\DOCUME~1\User\LOCALS~1\Temp
SESSIONNAME                    Console
PATHEXT                        .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;. . .
USERDOMAIN                     POPOV
PROCESSOR_ARCHITECTURE         x86
SystemDrive                    C:
. . .
```

Для получения значения определенной переменной среды нужно перед ее именем указать префикс env: (в оболочке cmd.exe для этой цели переменную нужно было заключать в знаки процента %). Например, следующая команда выведет на экран путь к корневому каталогу операционной системы, который хранится в переменной среды SystemRoot:

```
PS C:\> $env:SystemRoot
C:\WINDOWS.1
```

Изменять значения переменных среды в оболочке PowerShell можно при помощи следующего синтаксического выражения:

```
$env:имя_переменной = "новое_значение"
```

При этом следует иметь в виду, что изменения влияют только на текущий сеанс работы (аналогичным образом обстоит дело с командой `set` оболочки cmd.exe и с командой `setenv` в оболочках Unix-систем). Чтобы изменения стали постоянными, необходимо устанавливать их значения в системном реестре.

Массивы в PowerShell

В отличие от многих языков программирования, в PowerShell не нужно с помощью каких-либо специальных символов указывать начало массива или его конец, а также предварительно объявлять массив.

Для создания и инициализации массива можно просто присвоить значения его элементам. Значения, добавляемые в массив, разделяются запятой и отделяются от имени переменной (имени массива) оператором присваивания. Например, следующая команда создаст массив `$a` из трех элементов:

```
PS C:\> $a=1,2,3
PS C:\> $a
1
2
3
```

Можно также создать и инициализировать массив, используя оператор диапазона (..). Например, чтобы создать и инициализировать массив `$b`, содержащий значения от 10 до 14, можно выполнить следующую команду:

```
PS C:\> $b=10..14
```

В результате массив `$b` будет содержать пять значений:

```
PS C:\> $b
10
11
12
13
14
```

Обращение к элементам массива

Как мы уже видели, для отображения всех элементов массива нужно просто ввести его имя.

Длина массива (количество элементов) хранится в свойстве `Length`:

```
PS C:\> $a.Length
3
```

Для обращения к определенному элементу массива нужно указать его порядковый номер (индекс) в квадратных скобках после имени переменной. При этом следует иметь в виду, что нумерация элементов в массиве всегда начинается с нуля, поэтому для получения значения первого элемента нужно выполнить следующую команду:

```
PS C:\> $a[0]
1
```

В качестве индекса можно указывать и отрицательные значения, при этом отсчет будет вестись с конца массива. Например, индекс -1 будет соответствовать последнему элементу массива:

```
PS C:\> $a[-1]
3
```

Чтобы отобразить подмножество всех значений в массиве, можно применять *оператор диапазона*. Например, чтобы отобразить элементы с индексами от 1 до 2, введите:

```
PS C:\> $a[1..2]
2
3
```

В операторе диапазона можно использовать свойство `Length`. Например, для отображения элементов от индекса 1 до конца массива (последний элемент массива имеет индекс `Length-1`) можно выполнить следующую команду:

```
PS C:\> $a[1..($a.Length-1)]
2
3
```

Для изменения элемента массива нужно присвоить новое значение элементу с соответствующим индексом:

```
PS C:\> $a[0]=5
PS C:\> $a[1]=3.14
PS C:\> $a[2]="привет"
PS C:\> $a
5
3.14
привет
```

Операции с массивом

Последний пример показывает, что по умолчанию массивы PowerShell могут содержать элементы разных типов, то есть являются *полиморфными*. Определим тип нашего массива `$a`:

```
PS C:\> $a.GetType().FullName
System.Object[]
```

Итак, переменная $a имеет тип "массив элементов типа System.Object". Можно создать массив с жестко заданным типом, то есть такой массив, который содержит элементы только одного типа. Для этого, как и в случае с обычными скалярными переменными, необходимо указать нужный тип в квадратных скобках перед именем переменной (см. табл. 6.4). Например, следующая команда создаст массив 32-разрядных целых чисел:

```
PS C:\> [int[]]$a=1,2,3,4
```

Если попытаться записать в данный массив значение, которое нельзя преобразовать к целому типу, то возникнет ошибка:

```
PS C:\> $a[0]="aaa"
Ошибка при назначении массива для [0]: Не удается преобразовать
значение "aaa" в тип "System.Int32". Ошибка: "Input string was not
in a correct format.".
В строка:1 знак:4
+ $a[0 <<<< ]="aaa"
```

При попытке обратиться к элементу, выходящему за границы массива, возникнет ошибка. Например:

```
PS C:\> $a.Length
4
PS C:\> $a[4]=5
Произошла ошибка при назначении массива, так как индекс "4" находится вне
пределов допустимого диапазона.
В строка:1 знак:4
+ $a[4 <<<< ]=5
```

Подобные ошибки связаны с тем, что массивы PowerShell базируются на .NET-массивах, имеющих фиксированную длину. Несмотря на это, имеется способ увеличения длины массива. Для этого можно воспользоваться оператором конкатенации + или +=. Например, следующая команда добавит к массиву $a два новых элемента со значениями 5 и 6:

```
PS C:\> $a
1
2
3
4
PS C:\> $a+=5,6
PS C:\> $a
1
```

```
2
3
4
5
6
PS C:\>
```

При выполнении оператора += происходит следующее:

PowerShell создает новый массив, размер которого достаточен для помещения в него всех элементов.

Первоначальное содержимое массива копируется в новый массив.

Новые элементы копируются в конец нового массива.

Таким образом, мы на самом деле не добавляем новый элемент к массиву, а создаем новый массив большего размера.

Удалить элемент из массива не так просто, однако можно создать новый массив и скопировать в него все элементы кроме ненужного. Например, следующая команда создаст массив `$b`, содержащий все элементы массива `$a` кроме значения с индексом 2:

```
PS C:\> $b = $a[0,1 + 3..($a.Length-1)]
PS C:\> $b
1
2
4
5
6
PS C:\>
```

Можно объединить два массива в один с помощью оператора конкатенации +. Например:

```
PS C:\> $x=1,2
PS C:\> $y=3,4
PS C:\> $z=$x+$y
PS C:\> $z
1
2
3
4
PS C:\>
```

Следует иметь в виду, что обычный оператор присваивания (=) действует на массивы *по ссылке*. Например, создадим массив `$a` из двух элементов и присвоим этот массив переменной `$b`:

```
PS C:\> $a=1,2
PS C:\> $b=$a
PS C:\> $b
1
2
```

Теперь изменим значение первого элемента массива `$a` и посмотрим, что произойдет с массивом `$b`:

```
PS C:\> $a[0]="Новое значение"
PS C:\> $b
Новое значение
2
```

Как видите, содержимое массива `$b` также изменилось, так как переменная `$b` указывает на тот же объект, что и переменная `$a`.

Для удаления массива можно воспользоваться командлетом `Remove-Item` (псевдоним `del`) и удалить переменную, содержащую нужный массив, с виртуального диска `variable:`. Например:

```
PS C:\> $a
1
2
3
4
5
6
PS C:\> del variable:a
PS C:\> $a
PS C:\>
```

Хэш-таблицы (ассоциативные массивы)

Кроме обычных массивов в PowerShell поддерживаются так называемые *ассоциативные массивы* (иногда их также называют словарями) — структуры для хранения коллекции ключей и их значений, связанных попарно.

Например, можно использовать фамилию человека как ключ, а его дату рождения как значение. Ассоциативный массив обеспечивает структуру для хранения коллекции имен и дат рождения, где каждому имени сопоставлена дата рождения. Визуально массив ассоциированных значений можно представить как таблицу, состоящую из двух столбцов, где первый столбец является ключом, а второй — значением.

Ассоциативные массивы похожи на обычные массивы в PowerShell, но вместо обращения к содержимому массива по индексу, к элементам данных ассоциативного массива можно обращаться по ключу. Используя этот ключ, PowerShell возвращает соответствующее значение из ассоциативного массива.

Для хранения содержимого ассоциативного массива в PowerShell используется специальный тип данных — *хэш-таблица*, поскольку данная структура данных обеспечивает быстрый механизм поиска. Это очень важно, поскольку основным назначением ассоциативного массива является обеспечение эффективного механизма поиска.

> ***Замечание***
>
> В дальнейшем мы будем пользоваться терминами хэш-таблица и ассоциативный массив как синонимами.

В отличие от обычного массива для объявления и инициализации хэш-таблиц используются специальные литералы:

```
$имя_массива = @{ключ1 = элемент1; ключ2 = элемент2; ...}
```

Итак, каждому значению хэш-таблицы нужно присвоить метку (ключ), перед перечислением содержимого массива нужно поставить символы `@{`, а завершить перечисление элементов символом `}`. Ключи и значения разделяются знаком равенства (`=`), пары "ключ-значение" разделяются между собой точкой с запятой (`;`).

Создадим, например, хэш-таблицу, в которой будут храниться данные об одном человеке (ассоциативный массив с тремя элементами):

```
PS C:\> $user=@{Фамилия="Попов"; Имя="Андрей"; Телефон="55-55-55"}
PS C:\> $user

Name                           Value
----                           -----
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55
```

Теперь нужно научиться обращаться к его элементам. В PowerShell доступ к хэш-таблицам возможен двумя способами: с использованием нотации свойств или нотации массивов. Обращение с использованием нотации свойств выглядит следующим образом:

```
PS C:\> $user.Фамилия
Попов
PS C:\> $user.Имя
Андрей
```

При данном подходе хэш-таблица рассматривается как объект: вы указываете имя нужного свойства и получаете соответствующее значение. Обращение к ассоциативному массиву с использованием нотации массивов происходит так:

```
PS C:\> $user["Фамилия"]
Попов
PS C:\> $user["Фамилия", "Имя"]
Попов
Андрей
```

При работе с хэш-таблицей как с массивом можно получать значения сразу для нескольких ключей.

Базовым типом для ассоциативных массивов PowerShell является тип `System.Collections.Hashtable`:

```
PS C:\> $user.GetType().FullName
System.Collections.Hashtable
```

В данном типе определены несколько свойств и методов, которые можно использовать (напомним, что полный список свойств и методов можно получить с помощью командлета `Get-Member`). Например, в свойствах `Keys` и `Values` хранятся все ключи и все значения, соответственно:

```
PS C:\> $user.Keys
Фамилия
Имя
Телефон
PS C:\> $user.Values
Попов
Андрей
55-55-55
```

Операции с хэш-таблицей

Давайте научимся добавлять элементы в хэш-таблицу, изменять и удалять их. Добавим в хэш-таблицу `$user` данные о возрасте человека и о городе, где он проживает:

```
PS C:\> $user.Возраст=33
PS C:\> $user

Name                           Value
----                           -----
Возраст                        33
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55

PS C:\> $user["Город"]="Саранск"
PS C:\> $user

Name                           Value
----                           -----
Возраст                        33
Город                          Саранск
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55
```

Таким образом, добавляются элементы в ассоциативный массив с помощью простого оператора присваивания с использованием нотации свойств или массивов. Теперь изменим значение уже имеющегося в массиве ключа. Делается это также с помощью оператора присваивания:

```
PS C:\> $user.Город="Москва"
PS C:\> $user

Name                           Value
----                           -----
Возраст                        33
Город                          Москва
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55
```

Для удаления элемента из ассоциативного массива используется метод `Remove()`:

```
PS C:\> $user.Remove("Возраст")
PS C:\> $user
```

```
Name                           Value
----                           -----
Город                          Москва
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55
```

Можно создать пустую хэш-таблицу, не указывая ни одной пары "ключ-значение", и затем заполнять ее последовательно по одному элементу:

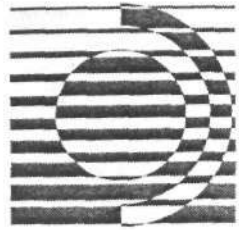
```
PS C:\> $a=@{}
PS C:\> $a
PS C:\> $a.one=1
PS C:\> $a.two=2
PS C:\> $a

Name                           Value
----                           -----
two                            2
one                            1
```

Как и в случае с обычными массивами, оператор присваивания действует на хэш-таблицы по ссылке. Например, после выполнения следующих команд переменные `$a` и `$b` будут указывать на один и тот же объект:

```
PS C:\> $a=@{one=1;two=2}
PS C:\> $b=$a
PS C:\> $b

Name                           Value
----                           -----
two                            2
one                            1
```

Поменяв значение одного из элементов в `$a`, мы получим то же изменение в `$b`:

```
PS C:\> $a.one=3
PS C:\> $b

Name                           Value
----                           -----
two                            2
one                            3
```



Глава 7

Операторы и управляющие инструкции

В языке PowerShell поддерживается много операторов, позволяющих выполнять различные действия. Одной из особенностей операторов PowerShell является их *полиморфизм*, то есть возможность применять один и тот же оператор к объектам разных типов. При этом отличие PowerShell от многих других объектно-ориентированных языков программирования состоит в том, что поведение операторов для основных типов данных (строки, числа, массивы и хэш-таблицы) реализуется непосредственно интерпретатором, а не с помощью того или иного метода объектов.

Арифметические операторы

Основные арифметические операторы, поддерживаемые в PowerShell, приведены в табл. 7.1.

Таблица 7.1. *Основные арифметические операторы в PowerShell*

Оператор	Описание	Пример	Результат
+	Складывает два значения	2+4 "aaa"+"bbb" 1,2,3+4,5	6 "aaabbb" 1,2,3,4,5
*	Перемножает два значения	2*4 "a"*3 1,2,3*2	8 "aaa" 1,2,3,1,2,3
-	*Вычитает одно значение из другого*	5-3	2
/	*Делит одно значение на другое*	6/3 7/4	2 1.75
%	Возвращает остаток при целочисленном делении одного значения на другое	7%4	3

С точки зрения полиморфного поведения наиболее интересными являются операторы сложения и умножения. Рассмотрим их более подробно.

Оператор сложения

Как уже упоминалось, поведение операторов + и * для чисел, строк, массивов и хэш-таблиц определяется самим интерпретатором PowerShell. В результате сложения или умножения двух чисел получается число. Результатом сложения (конкатенации) двух строк является строка. При сложении двух массивов создается новый массив, являющийся объединением складываемых массивов.

А что произойдет, если попытаться сложить объекты разных типов (например, число со строкой)? В этом случае поведение оператора будет определяться типом операнда, стоящего слева. Следует запомнить так называемое "*правило левой руки*": тип операнда, стоящего слева, задает тип результата действия оператора.

Если левый операнд является числом, то PowerShell попытается преобразовать правый операнд к числовому типу. Например:

```
PS C:\> 1+"12"
13
```

Как видите, строка "12" была преобразована к числу 12. Результат действия оператора сложения — число 13. Теперь обратный пример, когда левый операнд является строкой:

```
PS C:\> "1"+12
112
```

Здесь число 12 преобразуется к строке "12" и в результате конкатенации возвращается строка "112".

Если правый операнд нельзя преобразовать к типу левого операнда, то возникнет ошибка:

```
PS C:\> 1+"a"
Не удается преобразовать значение "a" в тип "System.Int32".
Ошибка: "Input string was not in a correct format."
В строка:1 знак:3
+ 1+" <<<< a"
```

Если операнд, стоящий слева от оператора сложения, является массивом, то стоящий справа операнд добавляется к этому массиву. При этом создается новый массив типа `[object[]]`, в который копируется содержимое операндов (это связано с тем, что размерность .NET-массивов фиксирована). В процессе

создания нового массива все ограничения на типы складываемых массивов будут утрачены. Рассмотрим пример. Создадим сначала массив целых чисел:

```
PS C:\> $a = [int[]] (1,2,3,4)
PS C:\> $a.GetType().FullName
System.Int32[]
```

Если попробовать изменить значение элемента этого массива на какую-либо строку, то возникнет ошибка, так как элементами массива `$a` могут быть только целые числа:

```
PS C:\> $a[0]="aaa"
Ошибка при назначении массива для [0]: Не удается преобразовать
значение "aaa" в тип "System.Int32". Ошибка: "Input string was not
in a correct format.".
В строка:1 знак:4
+ $a[0 <<<< ]="aaa"
```

Добавим теперь к массиву `$a` еще один символьный элемент с помощью оператора сложения:

```
PS C:\> $a=$a+"abc"
```

Вновь попробуем изменить значение первого элемента массива `$a`, записав в него символьную строку:

```
PS C:\> $a[0]="aaa"
PS C:\> $a
aaa
2
3
4
abc
```

Как видите, теперь ошибка не возникает. Это связано с изменением типа массива `$a`:

```
PS C:\> $a.GetType().FullName
System.Object[]
```

После увеличения массив `$a` получает тип `[object[]]`, и его элементами могут быть объекты любого типа.

Перейдем теперь к сложению хэш-таблиц. Как и в случае с обычными массивами, при сложении хэш-таблиц создается новый ассоциативный массив, в который копируются элементы из складываемых массивов. При этом оба операнда должны быть хэш-таблицами (никакое преобразование типов здесь не поддерживается). В новый массив сначала копируются элементы хэш-таблицы, стоящей слева от оператора сложения, а затем элементы хэш-таблицы,

стоящей справа. Если ключевое значение из правого операнда уже встречалось в левом, то при сложении возникнет ошибка.

Рассмотрим пример:

```
PS C:\> $left=@{a=1;b=2;c=3}
PS C:\> $right=@{d=4;e=5}
PS C:\> $sum=$left+$right
PS C:\> $sum

Name                           Value
----                           -----
d                              4
a                              1
b                              2
e                              5
c                              3
```

Новая результирующая хэш-таблица по-прежнему имеет тип `System.Collections.Hashtable`:

```
PS C:\> $sum.GetType().FullName
System.Collections.Hashtable
```

Оператор умножения

Оператор умножения может действовать на числа, строки и обычные массивы (операция умножения хэш-таблиц не определена). При этом справа от оператора умножения обязательно должно находиться число (в противном случае возникнет ошибка).

Если слева от оператора умножения расположено число, то операция умножения выполняется обычным образом:

```
PS C:\> 6*5
30
```

Если левым операндом является строка, то она повторяется указанное количество раз:

```
PS C:\> "abc"*3
abcabcabc
```

Если умножить строку на ноль, результатом будет пустая строка (тип `System.String`, длина равна 0):

```
PS C:\> "abc"*0

PS C:\> ("abc"*0).GetType().FullName
```

```
System.String
PS C:\> ("abc"*0).Length
0
```

Аналогичным образом оператор умножения действует на массивы:

```
PS C:\> $a=1,2,3
PS C:\> $a=$a*2
PS C:\> $a
1
2
3
1
2
3
```

Как и в случае оператора сложения, при умножении массива на число создается новый массив типа `[Object[]]` нужного размера и в него копируются элементы первоначального массива.

Операторы вычитания, деления и остатка от деления

Операторы вычитания (-), деления (/) и остатка от деления (%) в PowerShell определены только для чисел. Специального оператора целочисленного деления не предусмотрено: если делятся два целых числа, то результатом будет вещественное число (тип `System.Double`). Например:

```
PS C:\> 123/4
30.75
```

Если нужно округлить результат до ближайшего целого числа, то следует просто преобразовать его к типу `[int]`. Например:

```
PS C:\> [int](123/4)
31
```

При этом следует иметь в виду, что PowerShell при преобразовании вещественного числа в целое использует так называемое "округление Банкера": если число является полуцелым (0.5, 1.5, 2.5 и т. д.), то оно округляется до ближайшего четного числа. Таким образом, числа 1.5 и 2.5 округляются до 2, а 3.5 и 4.5 округляются до 4.

Если один из операндов является числом, а второй нет, то PowerShell пытается выполнить преобразование к числовому типу. Например:

```
PS C:\> "123"/4
30.75
```

```
PS C:\> 123/"4"
30.75
```

Если ни один из операндов не является числом, то возникает ошибка:

```
PS C:\> "123"/"4"
Произошла ошибка при вызове метода, так как [System.String] не
содержит метод с именем "op_Division".
В строка:1 знак:7
+ "123"/" <<<< 4"
```

Обратим внимание на последнее сообщение об ошибке. В PowerShell отсутствует собственное внутреннее определение данной операции, и система начинает искать определение оператора в типе данных левого операнда (оператору должен соответствовать метод с префиксом `op_`, например `op_Division` (деление), `op_Addition` (сложение), или `op_Subtraction` (вычитание)). В силу этого арифметические операторы срабатывают для некоторых нечисловых типов (например, для типа `System.Datetime`), в которых определены подобные методы.

Рассмотрим пример. Пусть в переменной `$d1` хранится объект типа `System.Datetime`, соответствующий дате 8 марта 2008 года, а в переменной `$d2` — объект, соответствующий дате 23 февраля 2008 года:

```
PS C:\> $d1=Get-Date -Year 2008 -Month 03 -Day 08
PS C:\> $d1.GetType().FullName
System.DateTime
PS C:\> $d2=Get-Date -Year 2008 -Month 02 -Day 23
```

Теперь мы можем вычесть из одной даты другую, определив, например, количество дней между ними:

```
PS C:\> ($d1-$d2).Days
13
```

Данная операция завершилась успешно, так как в типе `System.Datetime` определен метод `op_Subtraction`, соответствующий оператору вычитания.

Операторы присваивания

Наряду с простым оператором присваивания (=) в PowerShell поддерживаются *С-подобные составные операторы присваивания* (табл. 7.2).

Таблица 7.2. *Операторы присваивания в PowerShell*

Оператор	Пример	Аналог	Описание
=	$a=2		Присваивает переменной указанное значение
+=	$a+=3	$a=$a+3	Прибавляет указанное значение к текущему значению переменной, затем присваивает полученный результат данной переменной
-=	$a-=3	$a=$a-3	Отнимает указанное значение от текущего значения переменной, затем присваивает полученный результат данной переменной
*=	$a*=2	$a=$a*2	Умножает текущее значение переменной на указанное число, затем присваивает полученный результат данной переменной
/=	$a/=2	$a=$a/2	Делит текущее значение переменной на указанное число, затем присваивает полученный результат данной переменной
%=	$a%=2	$a=$a%2	Находит остаток от деления текущего значения переменной на указанное число, затем присваивает полученный результат данной переменной

В табл. 7.2 для каждого составного оператора присваивания приведен его аналог, составленный с помощью обычного оператора присваивания и соответствующего арифметического оператора. Следует иметь в виду, что для арифметических операторов здесь справедливы все правила и ограничения, описанные в предыдущем разделе. Например, оператор += можно применять к строкам:

```
PS C:\> $a="aa"
PS C:\> $a+="bb"
PS C:\> $a
aabb
```

Важной особенностью оператора присваивания в PowerShell является то, что с его помощью можно одной командой присвоить значения сразу нескольким переменным. При этом первый элемент присваиваемого значения будет присвоен первой переменной, второй элемент будет присвоен второй переменной, третий элемент — третьей переменной и т. д. Например, в результате

выполнения следующей команды переменной $a будет присвоено значение 1, а переменной $b значение 2:

```
PS C:\> $a,$b=1,2
PS C:\> $a
1
PS C:\> $b
2
```

Если присваиваемое значение содержит больше элементов, чем указано переменных, то оставшиеся значения будет присвоены последней переменной. Например, после выполнения следующей команды значением переменной $a будет являться число 1, а значением переменной $b — массив из чисел 2 и 3:

```
PS C:\> $a,$b=1,2,3
PS C:\> $a
1
PS C:\> $b
2
3
PS C:\> $b.GetType().FullName
System.Object[]
```

Операторы сравнения

Операторы сравнения позволяют сравнивать значения параметров в командах. При каждой операции сравнения создается условие, в зависимости от выполнения или невыполнения которого оператор принимает либо значение $True (истина), либо значение $False (ложь).

В PowerShell поддерживается несколько операторов сравнения, причем для каждого такого оператора определены версии, учитывающие и не учитывающие регистр букв. Основные операторы сравнения приведены в табл. 7.3.

Таблица 7.3. *Операторы сравнения в PowerShell*

Оператор	Значение	Пример (возвращается значение $true)
-eq -ceq -ieq	равно	10 -eq 10
-ne -cne -ine	не равно	9 -ne 10
-lt -clt -ilt	меньше	3 -lt 4
-le -cle -ile	меньше или равно	3 -le 4

Таблица 7.3 (окончание)

Оператор	Значение	Пример (возвращается значение $true)
-gt -cgt -igt	больше	4 -gt 3
-ge -cge -ige	больше или равно	4 -ge 3
-contains -ccontains -icintains	содержит	1,2,3 -contains 1
-notcontains -cnotcontains -inotcontains	не содержит	1,2,3 -notcontains 4

Базовый вариант операторов сравнения (-eq, -ne, -lt и т. д.) по умолчанию не учитывает регистр букв. Если оператор начинается с буквы "c" (-ceq, -cne, -clt и т. д.), то при сравнении регистр букв будет приниматься во внимание. Если оператор начинается с буквы "i" (-ieq, -ine, -ilt и т. д.), то регистр букв при сравнении не учитывается.

> ***Замечание***
>
> В PowerShell для обозначения операторов сравнения не используются обычные символы ">" и "<", так как они зарезервированы для перенаправления ввода/вывода.

Как и в случае с арифметическими операторами, для операторов сравнения справедливо "правило левой руки" (определяющим является тип левого операнда). Если число сравнивается со строкой (число стоит слева от оператора сравнения), то строка преобразовывается в число и выполняется сравнение двух чисел. Если левый операнд является строкой, то правый операнд преобразуется к символьному типу и выполняется сравнение двух строк.

Рассмотрим несколько примеров. Сравнение числа с числом:

```
PS C:\> 01 -eq 001
True
```

Сравнение числа со строкой (строка "001" преобразуется в число 1):

```
PS C:\> 01 -eq "001"
True
```

Сравнение строки с числом (число 001 преобразуется в строку "001"):

```
PS C:\> "01" -eq 001
False
```

Если левым операндом является массив, то оператор сравнения будет возвращать те элементы этого массива, которые соответствуют правому операнду. Например:

```
PS C:\> 1,2,3,4,1,2,3 -eq 1
1
1
PS C:\> 1,2,3,4,1,2,3 -ge 3
3
4
3
```

Рассмотрим теперь более подробно операторы соответствия шаблону (`-like` и `-match`).

Операторы проверки на соответствие шаблону

Наряду с основными операторами сравнения в PowerShell имеются операторы проверки символьных строк на соответствие определенному шаблону. При этом поддерживаются два вида шаблонов: *выражения с подстановочными символами* и *регулярные выражения*.

Шаблоны с подстановочными символами

Ранее мы уже использовали подстановочные (шаблонные) символы (`*` и `?`) при использовании, скажем, командлета `dir`. Например, команда `dir *.doc` выводит все файлы в текущем каталоге, имеющие расширение `doc` и любое имя (шаблонный символ `*` заменяет любое количество любых символов).

В PowerShell поддерживаются четыре вида подстановочных символов (табл. 7.4) и несколько операторов, проверяющих строки на соответствие шаблонам с подстановочными символами (табл. 7.5).

Таблица 7.4. *Подстановочные символы в PowerShell*

Символ	Описание	Пример	Соответствует	Не соответствует
*	Любое количество произвольных символов	a*	a ab abc	bc babc
?	Один произвольный символ	a?b	acb a1b	a ab

Таблица 7.4 (окончание)

Символ	Описание	Пример	Соответствует	Не соответствует
[<*симв1*>-<*симв2*>]	Диапазон символов от <симв1> до <симв2>	a[b-d]c	abc acc	aac ac
[<*симв1*><*симв2*>...]	Любой символ из указанного набора	a[bc]c	abc acc	a adc

Таблица 7.5. Операторы проверки на соответствие шаблону (подстановочные символы)

Оператор	Описание	Пример (возвращается значение $True)
-like -clike -ilike	Сравнение на совпадение с учетом подстановочного символа в тексте	"file.doc" -like "f*.doc"
-notlike -cnotlike -inotlike	Сравнение на несовпадение с учетом подстановочного символа в тексте	"file.doc" -notlike "f*.rtf"

Как видите, пользоваться шаблонами с подстановочными символами очень просто, однако возможности подобных шаблонов ограничены. Если возникает необходимость проверки более сложных условий, то нужно воспользоваться шаблонами с регулярными выражениями.

Шаблоны с регулярными выражениями

Регулярные выражения обобщают и расширяют концепцию шаблонов с подстановочными символами. Для задания образца используются *литералы* и *метасимволы*. Каждый символ, который не имеет специального значения в регулярных выражениях, рассматривается как литерал и должен точно совпасть при поиске. Например, буквы и цифры являются литеральными символами. Метасимволы — это символы со специальным значением в регулярных выражениях (табл. 7.6).

Замечание

Поддержка регулярных выражений в PowerShell реализована с помощью специальных классов платформы .NET. Язык регулярных выражений, поддерживаемый этими классами, обладает очень мощными возможностями, однако подробное рассмотрение данных вопросов выходит за рамки настоящей книги.

Дополнительную информацию о некоторых аспектах регулярных выражений можно найти во встроенной справке PowerShell (команда `Get-Help about_regular_expression`); более подробные справочные материалы имеются в библиотеке MSDN.

Таблица 7.6. *Некоторые метасимволы для регулярных выражений в PowerShell*

Символ	Описание	Пример	Соответствует	Не соответствует
. (точка)	Символ подстановки: соответствует любому символу	`a..`	`abc` `a34`	`a` `ab`

*	Повторитель: означает ноль или более предшествующих символов или классов символов	`a.*b`	`ab`
			`abc`
			`awerbc`
				`a`
				`bbc`

?	Повторитель: означает ноль или один предшествующий символ или класс символов	`a?b`	`b`
			`ab`
			`cb`
				`ac`
				`da`

^	Положение в строке: обозначает начало строки	`^ab`	`abc` `abcd`	`sab` `acb`
$	Положение в строке: обозначает конец строки	`ab$`	`sdab` `ab`	`abc` `abb`
[<*симв1*><*симв2*>...]	Класс символов: обозначает любой символ из указанного множества	`a[bc]c`	`abc` `acc`	`a` `adc`
[^<*симв1*><*симв2*>...]	Инвертированный класс: обозначает любой символ, не входящий в указанное множество	`a[^bc]c`	`adc` `a1c`	`abc` `acc`
[<*симв1*>-<*симв2*>]	Диапазон: обозначает любой символ из указанного промежутка	`a[b-d]c`	`abc` `acc`	`aac` `ac`
\<*метасимв*>	Исключение: определяет использование метасимвола как литерала	`\^ab`	`c^ab` `^ab`	`ab` `^ac`

В PowerShell с регулярными выражениями работают операторы -match, -notmatch и -replace, а также их варианты (табл. 7.7).

Таблица 7.7. *Операторы, работающие с регулярными выражениями*

Оператор	Описание	Пример	Результат
-match -cmatch -imatch	Сравнение на совпадение с учетом регулярных выражений в правом операнде	"книга" -match "ни" "нос" -match "[к-н]ос"	$True $True
-notmatch -cnotmatch -inotmatch	Сравнение на несовпадение с учетом регулярных выражений в правом операнде	"книга" -notmatch "^кн"	$False
-replace -creplace -ireplace	Замена или удаление символов в строке — левом операнде (эти операторы возвращают измененную строку). Если в качестве правого операнда указываются через запятую две подстроки, то первая из них соответствует фрагменту, который нужно изменить, а вторая — строке, которая будет вставлена в результате замены. Если в качестве правого операнда указана одна подстрока, то она соответствует фрагменту строки, который будет удален	"род" -replace "д", "т" "род" -replace "ро"	"рот" "д"

Логические операторы

Иногда внутри одной инструкции необходимо проверить сразу несколько условий. Операторы сравнения можно соединять друг с другом с помощью логических операторов, приведенных в табл. 7.8. При использовании логического оператора PowerShell проверяет каждое условие отдельно, а затем вычисляет значение инструкций целиком, связывая условия при помощи логических операторов.

Таблица 7.8. Логические операторы в PowerShell

Оператор	Значение	Пример (возвращается значение $True)
-and	логическое И	(10 -eq 10) -and (1 -eq 1)
-or	логическое ИЛИ	(9 -ne 10) -or (3 -eq 4)
-not	логическое НЕ	-not (3 -gt 4)
!	логическое НЕ	!(3 -gt 4)

Управляющие инструкции языка PowerShell

В языке PowerShell, как и в любом другом алгоритмическом языке, имеются элементы, позволяющие выполнить логическое сравнение и предпринять различные действия в зависимости от его результата или дающие возможность повторять одну или несколько команд снова и снова.

Инструкция *If ... ElseIf ... Else*

Логические сравнения лежат в основе практически всех алгоритмических языков программирования. В PowerShell при помощи инструкции `If` можно выполнять определенные блоки кода только в том случае, когда заданное условие имеет значение `$True` (истина). Также можно задать одно или несколько дополнительных условных блоков. Соответствующие им условия будут проверяться, если все предыдущие условия имели значение `$False`. Наконец, можно задать дополнительный блок кода, который будет выполняться в том случае, если ни одно из условий не имеет значения `$True`.

Синтаксис инструкции `If` в общем случае имеет следующий вид:

```
If (условие1)
   {блок_кода1}
[ElseIf (условие2)
    {блок_кода2}]
[Else
   {блок_кода3}]
```

При выполнении инструкции `If` проверяется истинность условного выражения *условие1*.

> **ЗАМЕЧАНИЕ**
>
> Условные выражения в PowerShell формируются, чаще всего, с помощью операторов сравнения и логических операторов, рассмотренных выше. Кроме того, важной особенностью языка является то, что в качестве условных выражений можно использовать конвейеры команд PowerShell.

Если *условие1* имеет значение $True, то выполняется *блок_кода1*, после чего PowerShell завершает выполнение инструкции If. Если *условие1* имеет значение $False, то PowerShell проверяет истинность условного выражения *условие2*. Если *условие2* имеет значение $True, то выполняется *блок_кода2*, после чего PowerShell завершает выполнение инструкции If. Если и *условие1*, и *условие2* имеют значение $False, то выполняется *блок_кода3*, и выполнение инструкции If завершается.

Приведем пример использования инструкции If в интерактивном режиме работы. Запишем сначала в переменную $a число 10:

```
PS C:\> $a=10
```

Сравним теперь значение переменной $a с числом 15:

```
PS C:\> If ($a -eq 15) {
>> 'Значение $a равно 15'
>> }
>> Else {'Значение $a не равно 15'}
>>
Значение $a не равно 15
```

Из данного примера также видно, что в оболочке PowerShell в интерактивном режиме можно выполнять инструкции, состоящие из нескольких строк (это может оказаться очень кстати при отладке сценариев).

Цикл *While*

В PowerShell поддерживаются несколько видов циклов. Самый простой из них — цикл While, в котором команды выполняются до тех пор, пока проверяемое условие имеет значение $True.

Инструкция While имеет следующий синтаксис:

```
While (условие) {блок_команд}
```

При выполнении инструкции While оболочка PowerShell вычисляет раздел *условие* инструкции, прежде чем перейти к разделу *блок_команд*. Условие в инструкции принимает значения $True или $False. До тех пор, пока условие имеет значение $True, PowerShell повторяет выполнение раздела *блок_команд*.

> ***Замечание***
>
> Как и в инструкции If, в условном выражении цикла While может использоваться конвейер команд PowerShell.

Раздел *блок_команд* инструкции While содержит одну или несколько команд, которые выполняются каждый раз при входе в цикл и его повторении.

Например, следующая инструкция `While` отображает числа от 1 до 3, если переменная `$val` не была создана или была создана и инициализирована значением 0:

```
PS C:\> While($val -ne 3)
>> {
>>     $val++
>>     $val
>> }
>>
1
2
3
```

В данном примере условие (значение переменной `$val` не равно 3) имеет значение `$True`, пока `$val` равно 0, 1 или 2. При каждом повторении цикла значение `$val` увеличивается на 1 с использованием унарного оператора увеличения значения ++ (`$val++`). При последнем выполнении цикла значение `$val` становится равным 3. При этом проверяемое условие принимает значение `$False`, и цикл завершается.

Цикл *Do ... While*

Цикл `Do ... While` похож на цикл `While`, однако условие в нем проверяется не до блока команд, а после:

```
Do {блок_команд} While (условие)
```

Например:

```
PS C:\> $val=0
PS C:\> Do {$val++; $val} While ($val -ne 3)
1
2
3
```

Цикл *For*

Инструкция `For` в PowerShell реализует еще один тип циклов — цикл со счетчиком. Обычно цикл `For` применяется для прохождения по массиву и выполнения определенных действий с каждым из его элементов. В PowerShell инструкция `For` используется не так часто, как в других языках программирования, так как коллекции объектов обычно удобнее обрабатывать с помощью инструкции `ForEach`. Однако если необходимо знать, с каким

именно элементом коллекции или массива мы работаем на данной итерации, то цикл `For` может помочь.

Синтаксис инструкции `For`:

```
For (инициация; условие; повторение) {блок_команд}
```

Составные части цикла `For` имеют следующий смысл:

- *инициация* — это одна или несколько разделяемых запятыми команд, выполняемых перед началом цикла. Эта часть цикла обычно используется для создания и присвоения начального значения переменной, которая будет затем основой для проверяемого условия в следующей части инструкции `For`;
- *условие* — часть инструкции `For`, которая может принимать логическое значение `$True` или `$False`. PowerShell проводит оценку условия при каждой итерации цикла `For`. Если результатом этой оценки является `$True`, то выполняются команды в блоке *блок_команд*, после чего проводится новая оценка условия инструкции. Если результатом оценки условия вновь становится `$True`, команды блока *блок_команд* выполняются вновь и т. д., пока результатом проверки условия не станет `$False`;
- *повторение* — это одна или несколько разделяемых запятыми команд, выполняемых при каждом повторении цикла. Данная часть цикла обычно используется для изменения переменной, проверяемой внутри условия. Эти команды выполняются после основного блока команд, но перед новой проверкой условия;
- *блок_команд* — это набор из одной или нескольких команд, выполняющихся при вхождении в цикл или при его повторении. Содержимое блока *блок_команд* заключается в фигурные скобки.

Классический пример:

```
PS C:\> For ($i=0; $i -lt 5; $i++) { $i }
0
1
2
3
4
```

Цикл *ForEach*

Инструкция `ForEach` позволяет последовательно перебирать элементы коллекций. Самым простым и наиболее часто используемым типом коллекции, по которой производится перемещение, является массив. Обычно в цикле `ForEach` одна или несколько команд выполняются на каждом элементе массива.

Особенностью цикла ForEach является то, что его синтаксис и работа зависят от того, где расположена инструкция ForEach: вне конвейера команд или внутри конвейера.

Инструкция *ForEach* вне конвейера команд

В этом случае синтаксис цикла ForEach имеет следующий вид:

```
ForEach ($элемент in $коллекция) {блок_команд}
```

В круглых скобках указывается коллекция, по которой производится итерация. При выполнении цикла ForEach система автоматически создает переменную *$элемент*. Перед каждой итерацией в цикле этой переменной присваивается значение очередного элемента в коллекции. В разделе *блок_команд* содержатся команды, выполняемые на каждом элементе коллекции.

Например, цикл ForEach в следующем примере отображает значения в массиве с именем $letterArray:

```
PS C:\> $letterArray = "a","b","c","d"
PS C:\> ForEach ($letter in $letterArray) {Write-Host $letter}
a
b
c
d
```

В первой команде здесь создается массив $letterArray, в который записываются четыре элемента: символы "a", "b", "c" и "d". При первом выполнении инструкции ForEach переменной $letter присваивается значение, равное первому элементу в $letterArray ("a"), затем используется командлет Write-Host для отображения переменной $letter. При следующей итерации цикла переменной $letter присваивается значение "b" и т. д. После того как будут перебраны все элементы массива $letterArray, произойдет выход из цикла.

Инструкция ForEach может также использоваться совместно с командлетами, возвращающими коллекции элементов. Например:

```
PS C:\> $l = 0; ForEach ($f in dir *.txt) { $l += $f.Length }
PS C:\> $l
2690555
```

Здесь сначала создается и обнуляется переменная $l, затем в цикле ForEach с помощью командлета dir формируется коллекция файлов с расширением txt, находящихся в текущем каталоге. В инструкции ForEach перебираются все элементы этой коллекции, на каждом шаге к текущему элементу (соответствующему файлу) можно обратиться с помощью переменной $f. В блоке

команд цикла ForEach к текущему значению переменной $l добавляется значение поля Length (размер файла) переменной $f. В результате выполнения данного цикла в переменной $l будет храниться суммарный размер файлов в текущем каталоге, которые имеют расширение txt.

Инструкция *ForEach* внутри конвейера команд

Если инструкция ForEach появляется внутри конвейера команд, то PowerShell использует псевдоним ForEach, соответствующий командлету ForEach-Object. То есть в этом случае фактически выполняется командлет ForEach-Object, и уже не нужно указывать часть инструкции (*$элемент* in *$коллекция*), так как элементы коллекции блоку команд предоставляет предыдущая команда в конвейере.

Синтаксис инструкции ForEach, применяемой внутри конвейера команд, в простейшем случае выглядит следующим образом:

```
команда | ForEach {блок_команд}
```

Рассмотренный выше пример с подсчетом суммарного размера текстовых файлов из текущего каталога для данного варианта инструкции ForEach примет следующий вид:

```
PS C:\> $l = 0; dir *.txt | ForEach { $l += $_.Length }
PS C:\> $l
2690555
```

> ***Замечание***
>
> Напомним, что специальная переменная $_ используется в командлетах, производящих обработку элементов конвейера, для обращения к текущему объекту конвейера и извлечения его свойств.

В общем случае в псевдониме ForEach может указываться не один блок команд, а три: начальный блок команд, средний блок команд и конечный блок команд. Начальный и конечный блоки команд выполняются один раз, а средний блок команд выполняется каждый раз при очередной итерации по коллекции или массиву.

Синтаксис псевдонима ForEach, используемого в конвейере команд с начальным, средним и конечным блоками команд выглядит следующим образом:

```
команда | ForEach {начальный_блок_команд} {средний_блок_команд} {конечный_
блок_команд}
```

Для этого варианта инструкции ForEach наш пример можно записать следующим образом:

```
PS C:\> dir *.txt | ForEach {$l = 0} { $l += $_.length } {Write-Host $l}
2690555
```

Метки циклов, инструкции *Break* и *Continue*

Инструкция `Break` позволяет выйти из цикла любого типа, не дожидаясь окончания его итераций. Рассмотрим простой пример:

```
PS C:\> $i=0; While ($True) {If ($i++ -ge 3) { Break } $i }
1
2
3
```

Условием цикла `While` в данном случае является логическая константа `$True`, поэтому этот цикл не завершился бы никогда. Инструкция `Break`, которая срабатывает при достижении переменной `$i` значения 3, позволяет выйти из данного цикла.

Инструкция `Continue` осуществляет переход к следующей итерации цикла любого типа. Например:

```
PS C:\> For ($i=0; $i -le 5; $i++) { If ($i -ge 4) { Continue } $i }
0
1
2
3
```

В данном примере на экран выводятся только числа от 0 до 3, так как для значений переменной `$i`, больших либо равных 4, срабатывает инструкция `Continue`.

В языке PowerShell поддерживается возможность немедленного выхода или перехода к следующей итерации не только для одиночного цикла, но и для вложенных циклов. Для этого циклам присваиваются специальные метки, которые указываются в начале строки перед ключевым словом, задающим цикл того или иного типа. Такие метки затем можно использовать во вложенных циклах совместно с инструкциями `Break` или `Continue`, указывая, на какой именно цикл должны действовать данные инструкции. Рассмотрим простой пример:

```
PS C:\> :outer While ( $True ) {
>>     While ( $True ) {
>>         Break outer
>>     }
>> }
>>
```

Здесь инструкция `Break` осуществляет выход из внешнего цикла с меткой `outer`. Если бы метка не была указана, внешний цикл не завершился бы никогда.

Инструкция *Switch*

Инструкция `Switch`, объединяющая несколько проверок условий внутри одной конструкции, имеется во многих языках программирования. Однако в языке PowerShell данная инструкция обладает мощными дополнительными возможностями:

- она может использоваться как аналог цикла, проверяя значения не единственного элемента, а целого массива;
- она может проверять элементы на соответствие шаблону с подстановочными символами или регулярными выражениями;
- она может обрабатывать текстовые файлы, используя в качестве проверяемых элементов строки из файлов.

Виды проверок внутри *Switch*

Рассмотрим вначале самую простую форму инструкции `Switch`, когда одно скалярное выражение по очереди сопоставляется с несколькими условиями. *Например:*

```
PS C:\> $a = 3
PS C:\> Switch ($a) {
>>      1 {"Один"}
>>      2 {"Два"}
>>      3 {"Три"}
>>      4 {"Четыре"}
>> }
>>
Три
```

В данном примере значение переменной `$a` последовательно сравнивается с числами 1, 2, 3 и 4. При совпадении выполняется соответствующий блок кода, указанный в фигурных скобках (в нашем случае просто выводится строка).

Если для проверяемого значения справедливы несколько условий из списка, то будут выполнены все действия, сопоставленные этим условиям. Например:

```
PS C:\> $a = 3
PS C:\> Switch ($a) {
>>      1 {"Один"}
>>      2 {"Два"}
>>      3 {"Три"}
>>      4 {"Четыре"}
```

```
>>     3 {"Еще раз три"}
>> }
>>
Три
Еще раз три
```

Если нужно ограничиться только первым совпадением, то следует применить инструкцию `Break`:

```
PS C:\> $a = 3
PS C:\> Switch ($a) {
>>     1 {"Один"}
>>     2 {"Два"}
>>     3 {"Три"; Break}
>>     4 {"Четыре"}
>>     3 {"Еще раз три"}
>> }
>>
Три
```

В данном случае проверка условий внутри `Switch` прерывается после нахождения первого соответствия. Если же ни одно соответствие не найдено, то инструкция `Switch` не выполняет никаких действий:

```
PS C:\> Switch (3) {
>>     1 {"Один"}
>>     2 {"Два"}
>>     4 {"Четыре"}
>> }
>>
PS C:\>
```

С помощью ключевого слова `Default` можно задать действие по умолчанию, которое будет выполняться в том случае, когда не найдено ни одно соответствие. Например:

```
PS C:\> Switch (3) {
>>     1 {"Один"}
>>     2 {"Два"}
>>     Default {"Ни один, ни два"}
>> }
>>
Ни один, ни два
```

По умолчанию в инструкции Switch производится прямое сравнение с объектами, указанными в условии. Сравнение строк при этом производится без учета регистров символов, например:

```
PS C:\> Switch ('абв') {
>>      'абв' {"Первое совпадение"}
>>      'АБВ' {"Второе совпадение"}
>> }
>>
Первое совпадение
Второе совпадение
```

Если при сравнении следует учесть регистр символов, то нужно указать параметр -CaseSensitive:

```
PS C:\> Switch -CaseSensitive ('абв') {
>>      'абв' {"Первое совпадение"}
>>      'АБВ' {"Второе совпадение"}
>> }
>>
Первое совпадение
```

Кроме обычного сравнения можно проверять элементы на соответствие шаблону с подстановочными символами. Для этого используется переключатель -Wildcard, например:

```
PS C:\> Switch -Wildcard ('абв') {
>>      'а*' {"Начинается с а"}
>>      '*в' {"Оканчивается на в"}
>> }
>>
Начинается с а
Оканчивается на в
```

Проверяемый элемент (объект) доступен внутри инструкции Switch через специальную переменную $_ (напомним, что в переменной с таким названием сохраняется и текущий элемент, передаваемый по конвейеру от одного командлета другому). Например:

```
PS C:\> Switch -Wildcard ('абв') {
>>      'а*' {"$_ начинается с а"}
>>      '*в' {"$_ оканчивается на в"}
>> }
>>
абв начинается с а
абв оканчивается на в
```

Как видите, переменная $_ в строке в двойных кавычках заменяется строкой, равной проверяемому значению.

Переключатель -Regex позволяет проверять элементы на соответствие шаблону, содержащему регулярные выражения (напомним, что они обсуждались ранее в этой главе). Предыдущий пример можно записать и следующим образом:

```
PS C:\> Switch -Regex ('абв') {
>>      '^а' {"$_ начинается с а"}
>>      'в$' {"$_ оканчивается на в"}
>> }
>>
абв начинается с а
абв оканчивается на в
```

Кроме проверок на простое совпадение или соответствие шаблону, инструкция Switch позволяет производить более сложные проверки, указывая вместо шаблонов блоки кода на языке PowerShell. Проверяемое значение при этом вновь доступно через переменную $_. Рассмотрим пример:

```
PS C:\> Switch (10) {
>>      {$_ -gt 5} {"$_ больше 5"}
>>      {$_ -lt 20} {"$_ меньше 20"}
>>      10 {"$_ равно 10"}
>> }
>>
10 больше 5
10 меньше 20
10 равно 10
```

В данном случае выполняются все три проверяемых условия: значения первых двух выражений "10 -gt 5" и "10 -lt 20" равны $True, третье условие — обычное сравнение двух чисел на равенство (10 равно 10).

Проверка массива значений

До настоящего момента все значения, которые мы проверяли в инструкции Switch, были скалярными величинами. Язык PowerShell допускает использование в качестве проверяемого значения массивы элементов, причем массивы могут задаваться явно или получаться в результате выполнения какой-либо команды, скажем, путем считывания строк из текстового файла. Рассмотрим пример:

```
PS C:\> Switch (1,2,3,4,5,6) {
>>      {$_ % 3} {"$_ не делится на три"}
```

```
>>     Default {"$_ делится на три"}
>> }
>>
1 не делится на три
2 не делится на три
3 делится на три
4 не делится на три
5 не делится на три
6 делится на три
```

В данном случае массив целых чисел задается явным перечислением своих элементов. Все элементы массива по очереди проверяются внутри инструкции `Switch`: если остаток от деления текущего элемента на 3 не равен нулю, то выводится сообщение "$_ не делится на 3", где вместо `$_` подставляется, как обычно, значение проверяемого элемента. В противном случае выводится сообщение "$_ делится на 3".

Приведем еще один пример, когда массив проверяемых элементов является результатом выполнения команды PowerShell. Пусть нам нужно узнать количество файлов с расширениями txt и log, находящихся в системном каталоге Windows. Сначала обнуляем переменные-счетчики:

```
PS C:\> $txt=$log=0
```

Выполним теперь подходящую инструкцию `Switch`. Коллекцию файлов из системного каталога Windows получим с помощью команды `dir $env:SystemRoot` (напомним, что в переменной среды `SystemRoot` хранится путь к нужному каталогу). Файлы с расширениями txt и log будем отбирать, проверяя на соответствие шаблонам с подстановочными символами (*.txt и *.log), и в соответствующих блоках кода будем увеличивать значение переменных `$txt` и `$log`:

```
PS C:\> Switch -Wildcard (dir $env:SystemRoot) {
>>     *.txt {$txt++}
>>     *.log {$log++}
>> }
>>
```

Выведем теперь значения переменных `$txt` и `$log`:

```
PS C:\> "txt-файлы: $txt    log-файлы: $log"
txt-файлы: 21    log-файлы: 156
```

Для использования в качестве входного массива строк из определенного текстового файла нужно указать в инструкции `Switch` параметр `-File` и путь к нужному файлу. Рассмотрим пример. Пусть в системном каталоге Windows

имеется файл KB946627.log с протоколом установки очередного обновления операционной системы. Выведем с помощью инструкции `Switch` все строки из этого файла, содержащие подстроки "Source:" или "Destinantion:" (для этого используются параметры `-Wildcard` и `-File`, а строки из файла проверяются на соответствие шаблону с подстановочными символами):

```
PS C:\> Switch -Wildcard -File $env:SystemRoot\KB946627.log {
>>     *Source:* {$_}
>>     *Destination:* {$_}
>> }
>>
10.281: Source:C:\WINDOWS\system32\SET32.tmp (5.1.2600.3173)
10.281: Destination:C:\WINDOWS\system32\rpcrt4.dll (5.1.2600.2180)
10.281: Source:C:\WINDOWS\system32\SETDB.tmp (5.1.2600.3243)
10.281: Destination:C:\WINDOWS\system32\xpsp3res.dll (5.1.2600.3132)
10.281: Source:C:\WINDOWS\system32\SET9E.tmp (6.0.2900.3231)
10.281: Destination:C:\WINDOWS\system32\wininet.dll (6.0.2900.3132)
10.281: Source:C:\WINDOWS\system32\SET9F.tmp (6.0.2900.3231)
10.281: Destination:C:\WINDOWS\system32\urlmon.dll (6.0.2900.3132)
. . .
```

Глава 8

Функции, фильтры и сценарии

До настоящего момента мы работали преимущественно со стандартными скомпилированными командлетами, функциональность которых мы не можем изменить, так как их исходный код в оболочке PowerShell недоступен. Функции и сценарии — это два других типа команд PowerShell, которые можно создавать и изменять по своему усмотрению, пользуясь языком PowerShell.

Следует сразу обратить внимание, что функции в PowerShell имеют некоторые особенности по сравнению с функциями в традиционных языках программирования, так как PowerShell — это в первую очередь оболочка. В традиционных языках функция, как правило, является аналогом метода объекта, а в PowerShell функция — это команда. Отсюда различие в способах вызова функций и передачи им аргументов.

Обычно функции в традиционных языках возвращают единственное значение того или иного типа. "Значением" функции PowerShell может быть целый массив различных объектов, так как каждое вычисляемое в функции выражение помещает свой результат в выходной поток.

Кроме того, функции в PowerShell делятся на несколько типов согласно их поведению внутри конвейера команд.

Поэтому для грамотной работы в оболочке и написания корректных программ на языке PowerShell необходимо изучить вопросы, связанные с реализацией функций и сценариев в этой системе.

Функции в PowerShell

Напомним, что функция в PowerShell — это блок кода, имеющий название и находящийся в памяти до завершения текущего сеанса командной оболочки. Если функция определяется без формальных параметров, то для ее задания

достаточно указать ключевое слово Function, затем имя функции и список выражений, составляющих тело функции (данный список должен быть заключен в фигурные скобки). Например, создадим функцию MyFunc:

```
PS C:\> Function MyFunc{"Всем привет!"}
PS C:\>
```

Для вызова этой функции нужно просто ввести ее имя:

```
PS C:\> MyFunc
Всем привет!
```

Отметим, что во многих языках программирования при вызове функции после ее имени нужно указывать круглые скобки. В PowerShell этого делать нельзя:

```
PS C:\> MyFunc()
Выражение ожидается после '('.
В строка:1 знак:8
+ MyFunc() <<<<
```

Функции могут работать с аргументами, которые передаются им при запуске, причем поддерживаются два варианта обработки таких аргументов: с помощью переменной $Args и путем задания формальных параметров.

Обработка аргументов функций с помощью переменной *$Args*

Функция в PowerShell имеет доступ к аргументам, с которыми она была запущена, даже если при определении этой функции не были заданы формальные параметры. Все аргументы, с которыми была запущена функция, автоматически сохраняются в переменной $Args. Другими словами, в переменной $Args содержится массив, элементами которого являются параметры функции, указанные при ее запуске. Для примера добавим переменную $Args в нашу функцию MyFunc:

```
PS C:\> Function MyFunc{"Привет, $Args!"}
```

Так как переменная $Args помещена в строку в двойных кавычках, то при запуске функции значение этой переменной будет вычислено (расширено), и результат будет вставлен в строку. Вызовем функцию MyFunc с тремя параметрами:

```
PS C:\> MyFunc Андрей Сергей Иван
Привет, Андрей Сергей Иван!
```

Как видите, три указанных нами параметра (три элемента массива $Args) помещены в выходную строку и разделены между собой пробелами. Можно

изменить символ, разделяющий элементы массивов при их подстановке в расширяемые строки, переопределив значение специальной переменной $OFS:

```
PS C:\> Function MyFunc{
>> $OFS=","
>> "Привет, $Args!"}
>>
PS C:\> MyFunc Андрей Сергей Иван
Привет, Андрей,Сергей,Иван!
```

Обратите внимание, что в отличие от традиционных языков программирования, функции в PowerShell являются командами (это не методы объектов!), поэтому их аргументы указываются через пробел без дополнительных круглых скобок и выделения символьных строк кавычками. Чтобы наглядно убедиться в этом, запустим функцию MyFunc следующим образом:

```
PS C:\> MyFunc("Андрей","Сергей","Иван")
Привет, System.Object[]!
```

Напомним, что массивы в PowerShell задаются перечислением своих элементов, а круглые скобки, окружающие какое-либо выражение, означают, что это выражение должно быть вычислено. Поэтому ошибки при таком вызове функции не возникает, однако результат ее выполнения оказывается совсем другим, ведь в данном случае в функцию передаются не три символьных аргумента, а один аргумент, являющийся массивом из трех элементов!

Так как переменная $Args является массивом, то внутри функции можно обращаться к отдельным аргументам по их порядковому номеру (напомним, что нумерация элементов массивов начинается с нуля), а с помощью метода Count определять общее количество аргументов, переданных функции. Для примера создадим функцию SumArgs, которая будет сообщать о количестве своих аргументов и вычислять их сумму:

```
PS C:\> Function SumArgs{"Количество аргументов: $($Args.Count)"
>> $n=0
>> For($i=0; $i -lt $Args.Count; $i++) { $n+=$Args[$i] }
>> "Сумма аргументов: $n"}
>>
PS C:\>
```

Запустим функцию SumArgs с тремя числовыми аргументами:

```
PS C:\> SumArgs 1 2 3
Количество аргументов: 3
Сумма аргументов: 6
```

Помимо использования массива $Args в PowerShell поддерживается альтернативный подход к обработке аргументов функций — с помощью задания формальных параметров.

Формальные параметры функций

В PowerShell, как и в большинстве других языков программирования, при описании функции можно задать список формальных параметров, значения которых во время выполнения функции будут заменены значениями фактически переданных аргументов.

Список формальных параметров указывается в круглых скобках после имени функции. Определим, например, функцию Subtract для нахождения разности двух своих аргументов (уменьшаемому соответствует параметр $From, вычитаемому — параметр $Count):

```
PS C:\> Function Subtract ($From, $Count) {$From-$Count}
```

При вызове функции Subtract ее формальные параметры будут заменены фактическими аргументами, определяемыми либо по позиции в командной строке, либо по имени.

> ***Замечание***
>
> Еще раз напомним, что аргументы функции указываются так же, как параметры других команд: через пробел, без круглых скобок!

Например:

```
PS C:\> Subtract 10 2
8
```

В этом случае соответствие формальных параметров фактически переданным аргументам определяется по позиции: вместо первого параметра $From подставляется число 10, вместо второго параметра $Count подставляется число 2.

При указании аргументов можно использовать имена формальных параметров (порядок указания аргументов при этом становится несущественным), например:

```
PS C:\> Subtract -From 10 -Count 2
8
PS C:\> Subtract -Count 3 -From 5
2
```

Если дважды указать имя одного и того же параметра, то возникнет ошибка:

```
PS C:\> Subtract -From 10 -From 2
Subtract : Не удается привязать параметр, так как параметр "From"
```

```
указан более одного раза. Для указания множественных значений
параметров, допускающих множественные значения, используйте
синтаксис массива.  Например, "-parameter value1,value2,value3".
В строка:1 знак:24
+ Subtract -From 10 -From  <<<< 2
```

При вызове функции возможен и третий вариант задания аргументов, когда для некоторых задаются имена, а некоторые определяются по позиции в командной строке. При этом действует следующий алгоритм:

Все именованные аргументы сопоставляются соответствующим формальным параметрам и удаляются из списка аргументов.

Оставшиеся параметры (безымянные или имеющие имя, которому не соответствует ни один формальный параметр) сопоставляются формальным параметрам по позиции.

Например:

```
PS C:\> Subtract -From 10 2
8
PS C:\> Subtract -Count 2 10
8
```

> ***Замечание***
>
> *Без необходимости при вызове функций не следует смешивать именованные и "позиционные" аргументы — это поможет избежать возможных ошибок при сопоставлении формальных и фактических параметров.*

По умолчанию функции PowerShell, как и другие команды, ведут себя полиморфным образом. Например, определим функцию `Add`, складывающую два своих аргумента:

```
PS C:\> Function Add ($x, $y) {$x+$y}
```

Выполним эту функцию с аргументами-числами и аргументами-строками:

```
PS C:\> Add 2 3
5
PS C:\> Add "2" "3"
23
```

В первом случае функция `Add` возвращает число 5, а во втором — строку "23":

```
PS C:\> (Add 2 3).GetType().FullName
System.Int32
PS C:\> (Add "2" "3").GetType().FullName
System.String
```

При необходимости при объявлении функции можно явно задать тип формальных параметров. Например, определим функцию IntAdd, которая будет складывать два своих целочисленных аргумента:

```
PS C:\> Function IntAdd ([int] $x, [int] $y) {$x+$y}
```

Как и в предыдущем случае, вызовем данную функцию с аргументами-числами и с аргументами-строками:

```
PS C:\> IntAdd 2 3
5
PS C:\> IntAdd "2" "3"
5
```

Как видите, результат получается один и тот же, так как символьные аргументы преобразуются к целочисленному типу. Если преобразование выполнить не удается, то возникнет ошибка:

```
PS C:\> IntAdd "2a" "3"
IntAdd: Не удается преобразовать значение "2a" в тип
"System.Int32". Ошибка: "Input string was not in a correct
format."
В строка:1 знак:8
+ IntAdd  <<<< "2a" "3"
```

При вызове функции может быть указано большее количество фактических аргументов, чем было задано формальных параметров. При этом "лишние" аргументы будут помещены в массив $Args. Для иллюстрации рассмотрим функцию ShowArgs с двумя формальными параметрами:

```
PS C:\> Function ShowArgs ($x,$y) {
>> "Первый аргумент: $x"
>> "Второй аргумент: $y"
>> "Остальные аргументы: $Args"}
>>
```

Вызовем эту функцию с одним аргументом:

```
PS C:\> ShowArgs 1
Первый аргумент: 1
Второй аргумент:
Остальные аргументы:
```

В этом случае единственный аргумент сопоставляется с параметром $x, значение параметра $y устанавливается равным $Null, а массив $Args не содержит элементов ($Args.Length=0).

Запустим теперь функцию ShowArgs с двумя аргументами:

```
PS C:\> ShowArgs 1 2
Первый аргумент: 1
Второй аргумент: 2
Остальные аргументы:
```

Как видите, в данном случае параметры $x и $y получают значения 1 и 2, а массив $Args остается пустым.

Наконец, запустим функцию ShowArgs с тремя и четырьмя аргументами:

```
PS C:\> ShowArgs 1 2 3
Первый аргумент: 1
Второй аргумент: 2
Остальные аргументы: 3
PS C:\> ShowArgs 1 2 3 4
Первый аргумент: 1
Второй аргумент: 2
Остальные аргументы: 3 4
```

Дополнительные аргументы действительно сохраняются в массиве $Args.

При объявлении формальных параметров можно указать значения, которые будут принимать эти параметры по умолчанию (если явно не указан соответствующий фактический аргумент).

> ***Замечание***
>
> В качестве инициализирующих значений можно указывать как константы, так и выражения PowerShell.

Например, переопределим функцию Add как функцию с двумя формальными параметрами $x и $y, которые по умолчанию инициализируются числами 2 и 3:

```
PS C:\> Function Add ($x=2, $y=3) {$x+$y}
```

Если запустить эту функцию без аргументов, то оба параметра инициализируются значениями по умолчанию и функция возвращает число 5:

```
PS C:\> Add
5
```

Если запустить функцию Add с одним аргументом, то указанное значение присвоится параметру $x, а параметр $y вновь инициализируется значением по умолчанию ($y=3):

```
PS C:\> Add 5
8
```

При указании двух параметров функция Add вернет их сумму:

```
PS C:\> Add 6 6
12
```

До настоящего момента мы рассматривали фактические параметры функций двух типов: именованные (для которых указывается имя формального параметра и его значение) и позиционные (для которых указывается только значение аргумента, а соответствующий формальный параметр определяется по позиции данного аргумента в командной строке). Вспомним, что в командлетах PowerShell поддерживались аргументы третьего типа: параметры-переключатели, которые задавались только своим именем. Таким параметром является, например, переключатель -Recurse в командлете dir. Если этот переключатель указан, то команда dir действует не только на текущий каталог, но и на все его подкаталоги.

В функциях также можно использовать подобные параметры-переключатели, которые должны иметь тип SwitchParameter, имеющий псевдоним [Switch]. Значениями переключателя могут быть только $True или $False, поэтому инициализировать подобный формальный параметр не нужно.

Для примера определим функцию MyFunc с одним параметром-переключателем, определяющим поведение функции:

```
PS C:\> Function MyFunc ([Switch] $Recurse) {
>> if ($Recurse) {
>>   "Рекурсивный вариант функции"
>>   }
>> else {  "Обычный вариант функции"
>>   }
>> }
>>
```

Запустив функцию MyFunc без параметра, мы получим сообщение, что она работает в обычном режиме:

```
PS C:\> MyFunc
Обычный вариант функции
```

Если указан параметр -Recurse, то функция MyFunc будет работать в альтернативном режиме:

```
PS C:\> MyFunc -Recurse
Рекурсивный вариант функции
```

> ***Замечание***
>
> Не рекомендуется создавать функции, сценарии или командлеты, в которых параметр-переключатель по умолчанию инициализировался бы значением $True, так как это сильно усложняет логику их вызова.

Возвращаемые значения

В традиционных языках программирования функция обычно возвращает единственное значение определенного типа. В оболочке PowerShell дело обстоит иначе — здесь результаты всех выражений или конвейеров, вычисляемые внутри функции, направляются в выходной поток. Рассмотрим, например, функцию `MyFunc`, в которой вычисляются три числовых выражения:

```
PS C:\> Function MyFunc { 1+2; 3*3; 12/4}
```

Как видите, в этой функции явно не возвращается ни одно значение. Запустим функцию `MyFunc`:

```
PS C:\> MyFunc
3
9
3
```

На экран выводятся три строки с результатами вычислений. Теперь запустим данную функцию и сохраним результат ее работы в переменной `$Result`:

```
PS C:\> $Result=MyFunc
```

Проверим тип переменной `$Result`:

```
PS C:\> $Result.GetType().FullName
System.Object[]
```

Итак, переменная `$Result` является массивом объектов. Найдем длину этого массива и значения его элементов:

```
PS C:\> $Result.Length
3
PS C:\> $Result[0]
3
PS C:\> $Result[1]
9
PS C:\> $Result[2]
3
```

Итак, массив `$Result` содержит все значения, которые выводились на экран во время работы функции `MyFunc`. Такое поведение нужно учитывать при написании и отладке своих функций. Например, определим функцию `MyFunc` следующего вида:

```
PS C:\> Function MyFunc {
>> $name=Read-Host "Введите свое имя"
>> "Здравствуйте, $name!"
```

```
>> $name
>> }
>>
```

В данной функции запрашивается имя пользователя (ввод с клавиатуры осуществляется в переменную `$name` с помощью командлета `Read-Host`), выводится приветствие для этого пользователя, и значение переменной `$name` выводится в выходной поток. Запустим функцию `MyFunc`:

```
PS C:\> MyFunc
Введите свое имя: Андрей
Здравствуйте, Андрей!
Андрей
```

Проверим теперь, какие данные возвращает функция:

```
PS C:\> $Result=MyFunc
Введите свое имя: Андрей
PS C:\> $Result.Length
2
PS C:\> $Result
Здравствуйте, Андрей!
Андрей
```

Итак, кроме значения переменной `$name` в выходной поток попала и строка с приветствием, что, вероятно, не совпадает с нашим желанием. Поэтому вместо обычной строки в двойных кавычках для вывода символьной строки можно воспользоваться командлетом `Write-Host`, и тогда такая строка не будет добавляться в выходной поток:

```
PS C:\> Function MyFunc {
>> $name=Read-Host "Введите свое имя"
>> Write-Host "Здравствуйте, $name!"
>> $name
>> }
>>
PS C:\> $Result=MyFunc
Введите свое имя: Андрей
Здравствуйте, Андрей!
PS C:\> $Result.Length
6
PS C:\> $Result
Андрей
```

Как видите, командлет `Write-Host` выводит строку непосредственно на консоль, а не в выходной поток, поэтому в данном случае функция `MyFunc` возвращает единственную строку — значение переменной `$name`.

В языке PowerShell имеется инструкция `Return`, выполняющая немедленный выход из функции (аналог рассмотренной в *главе 7* инструкции `Break` для выхода из цикла). Например:

```
PS C:\> Function MyFunc {
>> "Эта строка выводится на экран"
>> Return
>> "Эта строка никогда не будет напечатана"
>> }
>>
PS C:\> MyFunc
Эта строка выводится на экран
PS C:\>
```

С помощью данной инструкции можно "возвратить" значение из функции, указав нужный объект после слова `Return`. Например, `Return 10*3` или `Return "Возвращаемая строка"`.

> ***Замечание***
>
> Инструкция `Return` включена в язык PowerShell скорее как дань традиции, так как аналогичные операторы есть практически во всех языках программирования. Однако следует помнить, что PowerShell является оболочкой, и имеет смысл говорить не о единственном значении, возвращаемом функцией, а о выходном потоке, в который функция помещает результаты вычисления выражений.

Функции внутри конвейера команд

Вся идеология PowerShell построена на применении конвейеров команд, и функции здесь не являются исключением: их тоже можно использовать внутри конвейеров. При этом для приема внутри функции входящего потока объектов, передаваемых от другой команды, служит переменная `$Input`. В этой переменной будет содержаться коллекция входящих объектов. Рассмотрим пример:

```
PS C:\> Function Sum {
>> $n=0
>> ForEach ($i in $Input) { $n+=$i }
>> $n
>> }
>>
PS C:\>
```

Здесь мы определили функцию Sum, которая будет суммировать элементы коллекции, поступившие к ней по конвейеру. Результат суммирования помещается в выходной поток как значение переменной $n. Передав функции Sum по конвейеру массив чисел, на выходе мы получим их сумму:

```
PS C:\> 1..10 | Sum
55
```

Перебирать элементы входного потока можно не только с помощью цикла ForEach, но и путем вызова метода MoveNext(), при этом обратиться к текущему элементу входного потока можно с помощью свойства Current. Функцию суммирования элементов входного потока можно переписать следующим образом:

```
PS C:\> Function Sum2 {
>> $n=0
>> While ($Input.MoveNext()) {
>>   $n+=$Input.Current
>>    }
>> $n
>> }
>>
```

Запустим функцию Sum2 и убедимся, что она выдает тот же результат, что и функция и Sum:

```
PS C:\> 1..10 | Sum2
55
```

Таким образом, написать функцию для работы внутри конвейера нетрудно (достаточно знать о назначении переменной $Input). Однако имеется один нюанс. При получении данных от предыдущей команды функция приостанавливает конвейер и запускается только один раз, когда сформирована вся коллекция входных элементов. Другими словами, обычная функция не поддерживает в полной мере механизм конвейеризации, когда элемент обрабатывается в конвейере, не дожидаясь генерации следующих элементов. Для более эффективной работы внутри конвейера функцию следует оформлять в виде *фильтра*.

Фильтры в PowerShell

Фильтр — это функция особого вида, которая, находясь внутри конвейера, запускается для каждого входящего элемента (обычные функции запускаются один раз для всей совокупности элементов, сформированной предыдущей командой конвейера).

Синтаксически фильтры отличаются от функций лишь тем, что вместо ключевого слова `Function` указывается слово `Filter`:

```
Filter имя_фильтра ( параметры ) { блок_кода }
```

Однако алгоритмы работы обычной функции и фильтра различаются, если они находятся внутри конвейера. В обычной функции доступ к входящим элементам конвейера осуществляется через коллекцию `$Input`. В фильтре же определена переменная `$_`, соответствующая текущему элементу конвейера, проходящему через данный фильтр.

Рассмотрим пример. Напишем фильтр, который будет удваивать числа, проходящие через него по конвейеру. Такой фильтр будет состоять всего из одного выражения:

```
PS C:\> Filter Double {$_*2}
```

Проверим работу данного фильтра:

```
PS C:\> 1..4 | Double
2
4
6
8
PS C:\> 1..4 | Double | Double
4
8
12
16
```

> ***Замечание***
>
> Функциональность фильтров очень похожа на функциональность командлета `ForEach-Object`, который можно считать безымянным фильтром.

Функции в качестве командлетов

Рассмотрев обычные функции и фильтры, можно сделать вывод, что они не позволяют в полной мере контролировать процесс обработки элементов, поступающих по конвейеру от предыдущей команды. Обычные функции прерывают конвейеризацию, дожидаясь формирования всех входных объектов, а в фильтрах отсутствует удобный механизм, позволяющий выполнять определенные действия перед обработкой первого или после обработки последнего элемента конвейера. При этом компилированные командлеты лишены упомянутых недостатков, и при их разработке можно задавать три типа действий:

- выполняемые до начала обработки первого входного элемента;

- выполняемые для каждого объекта входного потока;
- выполняемые после обработки последнего элемента, поступившего по конвейеру.

В PowerShell имеется возможность реализовать подобную функциональность командлетов с помощью пользовательских функций специального типа, имеющих три раздела: `Begin`, `Process` и `End`. В этих функциях можно инициализировать внутренние переменные перед началом работы с поступающими по конвейеру элементами, обработать каждый поступающий объект (не дожидаясь при этом формирования следующих элементов) и выполнить определенные завершающие действия после обработки последнего элемента конвейера. Общий синтаксис описания подобных функций имеет следующий вид:

```
Function имя_функции ( параметры ) {
  Begin {
    блок_кода_инициализация
  }
  Process {
    блок_кода_обработка_элемента
  }
  End {
    блок_кода_завершение
  }
}
```

Ключевое слово `Begin` определяет секцию, команды из которой будут выполнены перед началом обработки первого объекта конвейера. Команды и выражения из секции `Process` будут выполняться каждый раз при получении по конвейеру нового объекта (доступ к текущему элементу в данной секции осуществляется через переменную `$_`).

Рассмотрим пример. Определим функцию `MyCmdlet` со всеми тремя секциями:

```
PS C:\> Function MyCmdlet ($a) {
>> Begin {
>>   $n=0; "Инициализация: n=$n, a=$a"}
>> Process {
>>   $n++
>>   "Обработка конвейера: n=$n, a=$a, текущий объект = $_" }
>> End {"Завершение: n=$n, a=$a"}
>> }
```

В каждой из секций выводится информация о значении двух переменных: формальный параметр `$a` соответствует передаваемому в функцию аргументу,

переменная $n определяется в секции инициализации и последовательно увеличивается в секции обработки. Кроме этого в секции обработки выводится значение текущего объекта, поступившего по конвейеру (переменная $_).

Запустим описанную функцию, указав в качестве аргумента число 4, и передадим ей по конвейеру числа от 5 до 8:

```
PS C:\> 5..8 | MyCmdlet 4
Инициализация: n=0, a=4
Обработка конвейера: n=1, a=4, текущий объект = 5
Обработка конвейера: n=2, a=4, текущий объект = 6
Обработка конвейера: n=3, a=4, текущий объект = 7
Обработка конвейера: n=4, a=4, текущий объект = 8
Завершение: n=4, a=4
```

Как видите, аргумент функции (число 4) и переменная $n, созданная в секции инициализации, доступны во всех трех секциях. Запустим теперь функцию MyCmdlet без конвейера:

```
PS C:\> MyCmdlet 4
Инициализация: n=0, a=4
Обработка конвейера: n=1, a=4, текущий объект =
Завершение: n=1, a=4
```

Отсюда делаем вывод, что команды из секции Process запускаются один раз даже при отсутствии конвейера.

Сценарии PowerShell

До настоящего момента мы работали в оболочке PowerShell интерактивно, вводя команды и получая результат их выполнения. Если завтра потребуется выполнить те же команды, что и сегодня, нам придется вводить их заново. Поэтому часто используемые последовательности команд лучше сохранять во внешних сценариях, которые затем могут выполняться в пакетном режиме, то есть без участия человека.

Сценарии PowerShell представляют собой текстовые файлы с расширением ps1, в которых записан код (команды, операторы и другие конструкции) на языке PowerShell. В отличие от сценариев WSH и командных файлов интерпретатора cmd.exe, сценарии PowerShell можно писать поэтапно, непосредственно в самой оболочке, перенося затем готовый код во внешний текстовый файл. Такой подход значительно упрощает изучение языка и отладку сценариев, позволяя сразу видеть результат выполнения отдельных частей сценария.

Создание и запуск сценариев PowerShell

Создать файл со сценарием PowerShell можно разными способами:

- воспользоваться внешним текстовым редактором (например, стандартным Блокнотом Windows), вручную ввести нужные команды и сохранить файл с расширением ps1;
- выполнить нужные команды в оболочке PowerShell, скопировать их с консоли в буфер Windows и вставить в текстовый файл, открытый во внешнем текстовом редакторе (приемы работы из оболочки с буфером Windows обсуждались в *главе 2*), и затем сохранить полученный в результате файл с расширением ps1;
- работая в оболочке PowerShell, включить с помощью командлета `Start-Transcript` режим протоколирования команд, описанный в *главе 2*. В результате будет создан внешний файл со всеми запускавшимися в сеансе работы командами. Из этого файла можно скопировать нужные команды в другой текстовый файл и сохранить его с расширением ps1;
- находясь в оболочке PowerShell, оформить команды PowerShell в виде строк (обычных или автономных) и перенаправить с помощью символов > и >> эти строки во внешний файл с расширением ps1.

Воспользуемся последним из предложенных вариантов и создадим в каталоге c:\script простейший сценарий test.ps1, который будет состоять из единственной строки:

```
"Эта строка печатается из сценария PowerShell"
```

Для начала создадим каталог c:\script и сделаем его текущим:

```
PS C:\> md c:\script

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         23.03.2008     18:55            Script

PS C:\> cd c:\script
PS C:\Script>
```

Теперь запишем нужную строку (включая двойные кавычки) в файл test.ps1:

```
PS C:\Script> '"Эта строка печатается из сценария PowerShell"' > test.ps1
```

Попробуем теперь запустить наш сценарий. Напомним, что сценарии выполняются системой только в том случае, когда это разрешено текущей политикой

выполнения (мы рассматривали данный вопрос в *главе 4*, когда обсуждали загрузку пользовательских профилей, которые также являются сценариями PowerShell). По умолчанию действует политика `Restricted`, полностью запрещающая выполнение сценариев PowerShell. Это сделано из соображений безопасности, так как в сценариях может содержаться вредоносный код, который может повредить систему или несанкционированно воспользоваться пользовательскими данными.

Проверим активную политику выполнения с помощью командлета `Get-ExecutionPolicy`:

```
PS C:\Script> Get-ExecutionPolicy
RemoteSigned
```

Если в вашей системе действует более строгая политика безопасности (`Restricted` или `AllSigned`), то установите политику `RemoteSigned`, позволяющую выполнять неподписанные локальные сценарии:

```
PS C:\Script> Set-ExecutionPolicy RemoteSigned
```

Попробуем запустить сценарий, указав его имя:

```
PS C:\script> test.ps1
Условие "test.ps1" не распознано как командлет, функция, выполняемая
программа или файл сценария. Проверьте условие и повторите попытку.
В строка:1 знак:9
+ test.ps1 <<<<
```

Дело в том, что при запуске сценариев PowerShell путь к файлу с кодом нужно всегда указывать явно, даже если сценарий находится в текущем каталоге, так как это предотвращает возможный несанкционированный запуск другой исполняемой программы с аналогичным именем, находящейся, например, в системном каталоге. Поэтому запустим сценарий следующим образом (точка соответствует текущему каталогу):

```
PS C:\Script> .\test.ps1
Эта строка печатается из сценария PowerShell
```

Итак, в этом случае сценарий выполняется. Можно даже не указывать расширение ps1:

```
PS C:\Script> .\test
Эта строка печатается из сценария PowerShell
```

Естественно, путь можно было задать полностью:

```
PS C:\Script> C:\Script\test.ps1
Эта строка печатается из сценария PowerShell
PS C:\Script> C:\Script\test
Эта строка печатается из сценария PowerShell
```

Сценарии PowerShell можно запускать непосредственно из командной строки интерпретатора cmd.exe или с помощью пункта **Выполнить** меню **Пуск**. Для этого нужно указать полный путь к этому сценарию в качестве параметра программы powershell.exe (полный путь к powershell.exe можно не указывать), например:

```
C:\> powershell.exe c:\script\test.ps1
```

Напомним, что командные файлы интерпретатора cmd.exe, равно как и сценарии WSH на языках VBScript или Jscript, можно было запускать из Проводника Windows, просто щелкая мышью на значках этих сценариев. Со сценариями PowerShell этот метод не работает — если дважды щелкнуть мышью на значке сценария PowerShell, то он не запустится, а откроется для редактирования в Блокноте (с точки зрения безопасности это правильный подход, позволяющий предотвратить случайный запуск сценария).

Передача аргументов в сценарии

Разбор и обработка аргументов, передаваемых в сценарии, производится практически так же, как и в функциях (вообще, сценарий — это фактически функция, которая находится не в оперативной памяти, а на диске).

Аргументы указываются после имени сценария и разделяются между собой пробелами. Переменная `$Args` внутри сценария содержит массив, элементами которого являются аргументы функции, указанные при ее запуске. Для примера напишем сценарий SumArgs.ps1, который будет сообщать количество параметров, с которыми он запущен, и их сумму. Файл с исходным кодом создадим следующим образом: текст сценария оформим в виде автономной строки (подобные строки были описаны в *главе 6*) и перенаправим эту строку в файл SumArgs.ps1:

```
PS C:\Script> @'
>> "Количество аргументов: $($Args.count)"
>> $n=0
>> for($i=0; $i -lt $Args.count; $i++) { $n+=$Args[$i] }
>> "Сумма аргументов: $n"
>> '@ > SumArgs.ps1
>>
```

Запустим теперь сценарий SumArgs.ps1 с несколькими числовыми параметрами:

```
PS C:\Script> .\SumArgs 1 2 3
Количество аргументов: 3
Сумма аргументов: 6
```

Как видите, массив $Args в сценариях имеет тот же смысл и обрабатывается точно так же, как в функциях.

В сценариях можно определять формальные параметры, вместо которых во время выполнения будут подставляться фактические аргументы, переданные в сценарий. В функциях формальные параметры перечислялись в круглых скобках после имени, то есть вне тела функции. В сценариях так поступить нельзя, так как здесь все содержимое файла является телом сценария, поэтому формальные параметры задаются с помощью специальной инструкции Param. Эта инструкция должна быть самой первой командой в файле, предшествовать ей могут только пустые строки и комментарии. Для примера напишем сценарий add.ps1 с двумя формальными параметрами, который будет выводить сумму своих аргументов. Для создания сценария вновь воспользуемся автономной строкой, которую перенаправим в файл:

```
PS C:\Script> @'
>> Param ($x=2, $y=3)
>> $x+$y
>> '@ > add.ps1
>>
```

Запустим полученный сценарий с аргументами и без них:

```
PS C:\Script> .\add 10 20
30
PS C:\Script> .\add
5
```

Все работает, как и ожидалось: если аргументы не указаны, то внутри сценария используются значения по умолчанию.

Выход из сценариев

В обычном режиме выход из сценария, как и из функции, происходит после выполнения последней инструкции в нем. Напомним, что в функции можно было воспользоваться инструкцией Return для принудительного завершения работы. Аналог этой инструкции для сценариев — инструкция Exit, позволяющая завершить работу сценария и, при необходимости, вернуть определенный код возврата. Код возврата может анализироваться во внешних программах (например, командных файлах или сценариях WSH), запускающих сценарий PowerShell.

Рассмотрим пример. Создадим командный файл Call_PS.bat, в котором будет вызываться сценарий PowerShell, заданный строкой "'PoSH-script is working'; exit 10", и выводиться значение переменной среды ERRORLEVEL,

которое равно коду возврата последнего выполнявшегося процесса (листинг 8.1).

Листинг 8.1. Определение кода завершения сценария PowerShell (файл Call_PS.bat)

```
@echo off
echo Запускаем сценарий PowerShell...
powershell "'PoSH-script is working'; exit 10"
echo Код возврата процесса PowerShell: %ERRORLEVEL%
```

Запустив файл Call_PS.bat в оболочке cmd.exe, мы получим следующий результат:

```
C:\Script> call_ps.bat
Запускаем сценарий PowerShell...
PoSH-script is working
Код возврата процесса PowerShell: 10
```

Оформление сценариев. Комментарии

Предположим, что вы хотите с помощью PowerShell выполнить какие-либо разовые действия, которые не понадобится повторить в будущем (например, вам нужно создать определенный каталог на жестком диске и скопировать в него все файлы с расширением jpg с компакт-диска). Вряд ли в этом случае вы будете создавать отдельный сценарий, проще сразу набрать нужные команды непосредственно в оболочке.

Если же для решения определенной задачи вы все-таки решили написать сценарий, то, видимо, предполагаете неоднократно обращаться к нему в дальнейшем. Со временем у любого человека, использующего сценарии, скапливается целая их коллекция (какие-то пишутся самостоятельно, какие-то берутся у знакомых, из Интернета и т. д.). Чтобы быстро найти нужный сценарий, вспомнить, для чего он предназначен, каким образом работает и с какими параметрами должен запускаться, в сценарии обязательно нужно добавлять *комментарии*. Комментарии помогают пояснить назначение и логику работы сценария. Рекомендуется в обязательном порядке комментировать следующие части сценария:

- весь сценарий в целом (заголовок сценария);
- все функции, объявленные в сценарии;
- большие управляющие структуры, в том числе не входящие в функции;
- все строки сценария, требующие дополнительного пояснения.

В любом случае лучше, если в сценарии окажутся лишние комментарии, чем если их будет не хватать. В языке PowerShell комментарии начинаются со знака "#"; все символы, стоящие после этого знака, рассматриваются как часть комментария и не интерпретируются системой. Например:

```
PS C:\> # Вся строка - комментарий. Выражение 2+2 не вычисляется
PS C:\> 2+2 # Комментарий в конце строки. Выражение перед ним вычисляется
4
```

Желательно сразу приучить себя к добавлению во все сценарии заголовков, содержащих, например, следующую информацию:

- имя сценария;
- язык, на котором написан сценарий;
- дата создания сценария;
- автор сценария;
- назначение или краткое описание сценария;
- история изменений в сценарии.

Подобный заголовок, оформленный в виде комментариев, может выглядеть следующим образом:

```
##############################################
# Имя: Hello.ps1
# Язык: PoSH 1.0
# Дата создания: 01.05.2008
# Автор: Андрей Попов
# Описание: Вывод на экран приветствия
##############################################
```

Кроме наличия заголовка и комментариев в тексте, при чтении и разборе сценариев (особенно чужих) большое значение имеет стиль, в котором написаны эти сценарии, их структура. Два сценария, выполняющих одну и ту же задачу, могут восприниматься анализирующим их человеком по-разному. Приведем простой пример. Пусть первый сценарий hello1.ps1 состоит всего из одной строки:

```
"Привет всем!"
```

Второй сценарий hello2.ps1 создадим в следующем виде:

```
Function WriteHello {
  $s = "Привет всем!"
  $s
}
WriteHello
```

Таким образом, сценарий hello2.ps1 выводит на экран ту же строку, что сценарий hello1.ps1, однако для восприятия и анализа он намного сложнее.

Понятно, что структуры простейших сценариев, состоящих из одного или нескольких конвейеров команд, и сложных больших сценариев (с управляющими структурами, функциями, множеством операторов и т. п.) должны отличаться друг от друга. Небольшие сценарии можно писать в "плоском" формате, без использования функций. Сценарии большого объема лучше логически разделять на части, определять функции, отдельно инициализировать все использующиеся переменные. Достоинства и недостатки возможных вариантов структур сценариев кратко описаны в табл. 8.1.

Таблица 8.1. *Сравнение сценариев, имеющих различную структуру*

Структура	Достоинства	Недостатки
В сценарии не используются функции	Самый простой вариант написания сценариев. Хорошо подходит для небольших сценариев (до 100 строк), где отсутствует многократный вызов одного и того же кода	Тяжело выделить ключевые части сценария. Могут возникнуть трудности с созданием библиотек сценариев, так как в сценариях нет функций
В сценарии используются функции, которые расположены в тексте неупорядоченно	Никаких достоинств	Подобный сценарий тяжело читать и изменять
Сценарий организован следующим образом: 1. Раздел инициализации 2. Основное тело сценария 3. Все функции	Подобный сценарий легко читается. Он имеет ясную логическую структуру, поэтому нетрудно понять, как он работает	При разборе сценария постоянно приходится перемещаться по тексту от описания функций в основное тело сценария. В небольших сценариях, где отсутствует многократный вызов одного и того же кода, данную структуру лучше не использовать
Сценарий организован следующим образом: 1. Раздел инициализации 2. Все функции 3. Основное тело сценария	Подобная структура упрощает поиск основных операций, выполняемых в сценарии	При чтении кода приходится искать основное тело сценария, что может несколько осложнить разбор его логики

Таблица 8.1 *(окончание)*

Структура	Достоинства	Недостатки
Сценарий организован следующим образом: 1. Раздел инициализации 2. Вызов главной функции Main, содержащей основное тело сценария 3. Все остальные функции	Данный подход соответствует структуре приложений, принятой во многих языках программирования	При разборе сценария постоянно приходится перемещаться по тексту от описания функций в основное тело сценария

Независимо от выбранной структуры, при написании сценариев желательно придерживаться некоторых общих рекомендаций по форматированию текста, упрощающих восприятие и разбор сценария в дальнейшем.

- Переменным лучше давать осмысленные имена, указывающие на их назначение.
- Следует избегать использования слишком длинных строк. Если команда или конвейер команд получаются длинными, лучше разместить их на нескольких строках, ставя в конце каждой строки символ обратного апострофа (`).
- Перед ключевыми частями сценария (например, перед объявлением функций) лучше оставлять пустую строку.
- Комментарий лучше писать на отдельной строке перед выражением, к которому он относится, а не после этого выражения.

Соблюдение этих рекомендаций упрощает восприятие текста сценария и его дальнейший анализ.

Глава 9

Обработка ошибок и отладка

При применении сценариев для решения реальных администраторских задач (например, для управления учетными записями пользователей, резервного копирования информации на серверах, анализа журналов событий с целью выявления попыток несанкционированного доступа в систему и т. п.) весьма актуальной становится обработка возможных ошибок или исключительных ситуаций. Нужно быть уверенным, что определенная задача выполнена полностью, а в случае возникновения ошибки проанализировать и устранить ее. Язык PowerShell предлагает несколько механизмов обработки ошибок, которые мы рассмотрим подробнее в данной главе.

Кроме того, в оболочке PowerShell имеются средства отладки сценариев, помогающие находить ошибки в программном коде, а также анализировать ход выполнения сценариев и состояние системы при этом.

Давайте сначала посмотрим, с помощью каких объектов можно определить факт возникновения ошибки при выполнении команды PowerShell и какую информацию можно извлечь из этих ошибок.

Обработка ошибок

Возникновение ошибки при выполнении команды PowerShell приводит не только к выводу текстовых сообщений на экран, а еще и к автоматическому созданию настоящего объекта, в свойствах которого содержится полная информация о данной ошибке.

Один аспект обработки ошибок в PowerShell отличает эту систему от других языков программирования: ошибки здесь могут быть "*критическими*" (прерывающими выполнение команды) и "*некритическими*" (при их возникновении выполнение команды продолжается). Это связано с тем, что для обработки объектов в PowerShell используется модель конвейера команд.

При возникновении "некритической" ошибки информация о ней записывается в соответствующий объект, а текущая команда продолжает обрабатывать поступающие к ней по конвейеру объекты. Подобная ошибка может возникнуть, например, при копировании с помощью командлета множества файлов, один из которых оказался занят другим приложением. В таком случае логично продолжить копирование остальных файлов. Если же выполняется какая-то важная и логически неделимая операция, то может иметь смысл остановить ее после возникновения первой ошибки (тогда подобная ошибка будет "критической"). При этом одна и та же ошибка в одной ситуации должна считаться "критической", а в другой — "некритической". Поэтому в PowerShell встроен механизм, позволяющий явно задавать эти типы для ошибок.

При возникновении "некритических" ошибок информация о них помещается в объекты типа `ErrorRecord`, которые записываются в специальный поток ошибок. Рассмотрим, каким образом можно обращаться к данным объектам и какую информацию можно из них извлечь.

Объект ErrorRecord и поток ошибок

В PowerShell информация о возникающих ошибках записывается в поток ошибок, который по умолчанию отображается на экране, например:

```
PS C:\> dir "Несуществующий каталог"
Get-ChildItem : Не удается найти путь "C:\Несуществующий каталог",
так как он не существует.
В строка:1 знак:4
+ dir  <<<< "Несуществующий каталог"
```

Поток ошибок можно перенаправить в текстовый файл с помощью специального оператора `2>`, например:

```
PS C:\> dir "Несуществующий каталог" 2> err.txt
```

Проверим теперь содержимое файла err.txt:

```
PS C:\> Get-Content err.txt
Get-ChildItem : Не удается найти путь "C:\Несуществующий каталог",
так как он не существует.
В строка:1 знак:4
+ dir  <<<< "Несуществующий каталог" 2> err.txt
```

Как видите, в файл err.txt полностью записалось сообщение об ошибке, однако никакой дополнительной информации не появилось.

Объект ErrorRecord, создающийся при возникновении ошибки, можно сохранить в переменной. Для этого используется оператор 2>&1, который перенаправляет поток ошибок в стандартный выходной поток. Например:

```
PS C:\> $err = dir "Несуществующий каталог" 2>&1
PS C:\> $err.gettype().fullname
System.Management.Automation.ErrorRecord
```

Проверим с помощью командлета Get-Member, какие свойства имеет объект типа ErrorRecord:

```
PS C:\> $err | Get-Member -type property

   TypeName: System.Management.Automation.ErrorRecord

Name                 MemberType Definition
----                 ---------- ----------
CategoryInfo         Property   System.Management.Automation.Er...
ErrorDetails         Property   System.Management.Automation.Er...
Exception            Property   System.Exception Exception {get...
FullyQualifiedErrorId Property   System.String FullyQualifiedErr...
InvocationInfo       Property   System.Management.Automation.In...
TargetObject         Property   System.Object TargetObject {get...
```

Все эти свойства имеют отношение к возникающим ошибкам (табл. 9.1).

Таблица 9.1. *Свойства объекта ErrorRecord*

Свойство	Тип	Описание
CategoryInfo	System.Management.Automation.ErrorCategoryInfo	Категория ошибки
ErrorDetails	System.Management.Automation.ErrorDetails	Дополнительное описание ошибки (свойство ErrorDetails.Message), которое, в принципе, может быть пустым
Exception	System.Exception	Исключение .NET, соответствующее произошедшей ошибке
FullyQualifiedErrorId	System.String	Точное определение категории ошибки

Таблица 9.1 (окончание)

Свойство	Тип	Описание
InvocationInfo	System.Management.Automation.InvocationInfo	Объект, содержащий информацию о том, где именно произошла ошибка (обычно это имя сценария и номер строки в нем)
TargetObject	System.Object	Объект, над которым производились манипуляции во время возникновения ошибки. Данное свойство может быть пустым

Проверим содержимое переменной `$err` (напомним, что `fl` — это псевдоним командлета `Format-List`):

```
PS C:\> $err | fl * -Force

Exception             : System.Management.Automation.ItemNotFoundException:
                        Не удается найти путь "C:\Несуществующий каталог",
                        так как он не существует.
                           at System.Management.Automation.SessionState
                        Internal.GetChildItems(String path, Boolean recurse,
                        CmdletProviderContext context)
                           at System.Management.Automation.ChildItem
                        CmdletProviderIntrinsics.Get(String path,
                        Boolean recurse,CmdletProviderContext context)
                           at icrosoft.PowerShell.Commands.
                        GetChildItemCommand.ProcessRecord()
TargetObject          : C:\Несуществующий каталог
CategoryInfo          : ObjectNotFound: (C:\Несуществующий каталог:
                          String) [Get-ChildItem],ItemNotFoundException
FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                          GetChildItemCommand
ErrorDetails          :
InvocationInfo        : System.Management.Automation.InvocationInfo
```

В нашем примере ошибка относится к категории `PathNotFound`, свойство `TargetObject` содержит полный путь, который командлет `dir` использовал для поиска каталога или файла. Ошибка была вызвана исключением `ItemNotFoundException`.

Выясним, что же содержится в свойстве `InvocationInfo`:

```
PS C:\> $err.InvocationInfo

MyCommand        : Get-ChildItem
ScriptLineNumber : 1
OffsetInLine     : -2147483648
ScriptName       :
Line             : $err = dir "Несуществующий каталог" 2>&1
PositionMessage  :
                   В строка:1 знак:11
                   + $err = dir  <<<< "Несуществующий каталог" 2>&1
InvocationName   : dir
PipelineLength   : 1
PipelinePosition : 1
```

Как видите, свойство `ScriptName` здесь не заполнено, так как команда была введена непосредственно в интерактивном режиме оболочки, а не из сценария.

Итак, мы изучили структуру объекта `ErrorRecord` и узнали, какую информацию можно извлечь из его свойств. Напомним, что получили мы этот объект путем перенаправления потока ошибок в стандартный выходной поток в одной конкретной команде. Понятно, что для каждой выполняемой команды повторять эти манипуляции мы не сможем (ведь стандартный выходной поток может понадобиться для других целей) — значит, необходим более удобный механизм, позволяющий определять, возникла ли ошибка, и обращаться к объекту `ErrorRecord`, который соответствует той или иной ошибке. Подобные механизмы реализованы в PowerShell с помощью нескольких специальных переменных и параметров, к рассмотрению которых мы сейчас перейдем.

Сохранение объектов, соответствующих ошибкам

В PowerShell имеется специальная переменная `$Error`, которая содержит коллекцию (массив) объектов `ErrorRecord`, соответствующих ошибкам, возникавшим в текущем сеансе работы. Максимальное количество элементов в данной коллекции задается значением переменной `$MaximumErrorCount` (по умолчанию это 256):

```
PS C:\> $MaximumErrorCount
256
```

После заполнения массива `$Error` объекты для вновь возникающих ошибок будут заменять объекты, соответствующие старым ошибкам. Возникновение

каждой новой ошибки приводит к смещению элементов в массиве `$Error`: объект для последней ошибки хранится в первом элементе (`$Error[0]`), объект для предыдущей ошибки — во втором элементе (`$Error[1]`) и т. д.

Повторим еще раз ошибку из предыдущего раздела:

```
PS C:\> dir "Несуществующий каталог"
Get-ChildItem : Не удается найти путь "C:\Несуществующий каталог",
так как он не существует.
В строка:1 знак:4
+ dir  <<<< "Несуществующий каталог"
```

Теперь убедимся, что элемент `$Error[0]` имеет тип `ErrorRecord`, и выведем содержимое этого элемента:

```
PS C:\> $Error[0].GetType().FullName
System.Management.Automation.ErrorRecord
PS C:\> $Error[0]
Get-ChildItem : Не удается найти путь "C:\Несуществующий каталог",
так как он не существует.
В строка:1 знак:4
+ dir  <<<< "Несуществующий каталог"
```

Естественно, мы можем обращаться ко всем свойствам объекта `ErrorRecord`:

```
PS C:\> $Error[0].Exception
Не удается найти путь "C:\Несуществующий каталог", так как он
не существует.
PS C:\> $Error[0].InvocationInfo

MyCommand        : Get-ChildItem
ScriptLineNumber : 1
OffsetInLine     : -2147483648
ScriptName       :
Line             : dir "Несуществующий каталог"
PositionMessage  :
                   В строка:1 знак:4
                   + dir  <<<< "Несуществующий каталог"
InvocationName   : dir
PipelineLength   : 1
PipelinePosition : 1
PS C:\> $Error[0].TargetObject
C:\Несуществующий каталог
```

Для перехвата ошибок определенной команды без перенаправления ошибок в стандартный выходной поток можно воспользоваться стандартным параметром -ErrorVariable, который определен во всех командах PowerShell. Например, в следующей команде информация об ошибках будет записываться в переменную errs (обратите внимание, что знак $ перед именем переменной здесь указывать не нужно):

```
PS C:\> dir "Каталог 1", "Каталог 2" -ErrorVariable errs
Get-ChildItem : Не удается найти путь "C:\Каталог 1", так как он
не существует.
В строка:1 знак:4
+ dir  <<<< "Каталог 1", "Каталог 2" -ErrorVariable errs
Get-ChildItem : Не удается найти путь "C:\Каталог 2", так как он
не существует.
В строка:1 знак:4
+ dir  <<<< "Каталог 1", "Каталог 2" -ErrorVariable errs
```

При выполнении данной команды возникают две "некритические" ошибки, то есть переменная $errs должна быть массивом, содержащим два объекта ErrorRecord. Проверим это:

```
PS C:\> $errs.count
2
PS C:\> $errs[0].GetType().FullName
System.Management.Automation.ErrorRecord
PS C:\> $errs[1].GetType().FullName
System.Management.Automation.ErrorRecord
PS C:\> $errs[0].TargetObject
C:\Каталог 1
PS C:\> $errs[1].TargetObject
C:\Каталог 2
```

Как видите, при указании параметра -ErrorVariable сообщения о возникающих ошибках по-прежнему выводятся на экран. Для подавления этих сообщений нужно перенаправить поток ошибок на пустое устройство $Null, например:

```
PS C:\> dir "Каталог 1", "Каталог 2" -ErrorVariable errs 2> $Null
PS C:\>
```

На экран ничего не выводится, а в переменную $errs по-прежнему записываются два объекта:

```
PS C:\> $errs.count
2
```

При этом объекты, соответствующие ошибкам, записываются и в переменную $Error:

```
PS C:\> $Error[0]
Get-ChildItem : Не удается найти путь "C:\Каталог 2", так как он
не существует.
В строка:1 знак:4
+ dir  <<<< "Каталог 1", "Каталог 2" -ErrorVariable errs 2> $Null
PS C:\> $Error[1]
Get-ChildItem : Не удается найти путь "C:\Каталог 1", так как он
не существует.
В строка:1 знак:4
+ dir  <<<< "Каталог 1", "Каталог 2" -ErrorVariable errs 2> $Null
```

Мониторинг возникновения ошибок

При интерактивной работе в оболочке PowerShell мы узнаем о возникновении ошибки по сообщению на экране. После этого мы можем проанализировать объект $Error[0] и узнать всю информацию об ошибке. *Но как узнать о возникновении ошибки внутри сценария? Оказывается, в* PowerShell имеется *логическая переменная* $?, которая равна $True, если последняя выполняемая операция завершена успешно, и $False, если во время выполнения последней операции возникли какие-либо ошибки.

Например, если выполнить командлет Get-Item для заведомо существующего каталога, то значение переменной $? будет равно $True:

```
PS C:\> Get-Item c:\

    Каталог:

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d-r-hs        25.03.2008     18:06            C:\


PS C:\> $?
True
```

Если же выполнить этот же командлет для несуществующего каталога, то значение переменной $? будет равно $False:

```
PS C:\> Get-Item c:\АБВГГ
Get-Item : Не удается найти путь "C:\АБВГГ", так как он не существует.
В строка:1 знак:9
```

```
+ Get-Item  <<<< c:\АБВГ
PS C:\> $?
False
```

Для внешних команд Windows и сценариев PowerShell определено понятие кода возврата (напомним, что для сценариев PowerShell этот код можно установить с помощью инструкции `Exit`). В операционной системе код возврата последней команды доступен через переменную среды `%ERRORLEVEL%`; в оболочке PowerShell данный код возврата хранится в специальной переменной `$LASTEXITCODE`. При этом, если код возврата равен нулю, то переменной `$?` присваивается значение `$True`. Если же код возврата не равен нулю, то считается, что при выполнении данной команды произошла ошибка, и переменной `$?` присваивается значение `$False`.

В качестве примера выполним команду интерпретатора cmd.exe, которая устанавливает нулевой код возврата. Для этого можно запустить cmd.exe с ключом `/c` (выполнить команду и завершить работу интерпретатора) и указать для исполнения команду `exit 0`:

```
PS C:\> cmd /c exit 0
```

Проверим значения переменных `$LASTEXITCODE` и `$?`:

```
PS C:\> $LASTEXITCODE
0
PS C:\> $?
True
```

Теперь выполним команду интерпретатора cmd.exe с ненулевым кодом возврата (пусть, например, код возврата равен 10):

```
PS C:\> cmd /c exit 10
```

Вновь проверим переменные `$LASTEXITCODE` и `$?`:

```
PS C:\> $?
False
PS C:\> $LASTEXITCODE
10
```

Итак, мы убедились, что все работает правильно и переменными `$LASTEXITCODE` и `$?` вполне можно пользоваться для обнаружения ошибок при выполнении сценариев PowerShell.

Режимы обработок ошибок

В начале данной главы мы говорили о том, что ошибки в PowerShell делятся на "критические" (прерывающие выполнение команды) и "некритические" (при их возникновении выполнение команды продолжается). При этом в за-

висимости от ситуации "некритические" ошибки иногда может потребоваться рассматривать как "критические" и наоборот. Подобное изменение типа ошибок реализовано в PowerShell с помощью трех различных режимов обработки ошибок (табл. 9.2).

Таблица 9.2. *Режимы обработки ошибок в PowerShell*

Режим	Символьная константа	Описание
Продолжение выполнения команды	`"continue"`	Данный режим используется системой по умолчанию. Объект, соответствующий ошибке, записывается в выходной поток и добавляется в массив `$Error`. Переменная `$?` устанавливается в значение `$False`. На экран выводится сообщение об ошибке. Выполнение сценария продолжается со следующей строки
Продолжение выполнения команды без выдачи сообщений об ошибке	`"silentlycontinue"`	Если установлен данный режим, то при возникновении ошибки на экран не выводится никаких сообщений. Объект, соответствующий ошибке, записывается в выходной поток и добавляется в массив `$Error`. Переменная `$?` устанавливается в значение `$False`. Выполнение сценария продолжается со следующей строки
Прекращение выполнения команд	`"stop"`	Установка данного режима делает ошибки "критическими". Объект, соответствующий ошибке, не записывается в выходной поток, а генерирует исключение. При этом переменные `$?` и `$Error` по-прежнему обновляются. Выполнение сценария прекращается

Для установки нужно режима обработки ошибок нужно присвоить соответствующее значение переменной `$ErrorActionPreference`, например:

```
$ErrorActionPreference = "silentlycontinue"
```

В этом случае режим продолжения выполнения без выдачи сообщений об ошибке будет действовать на все команды. Если нужно переключить режим обработки ошибок только для одного командлета, то нужно при вызове

данного командлета указать соответствующую символьную константу в качестве значения параметра -ErrorAction (или сокращенно ea), например:

```
PS C:\> Get-Item "несуществующий файл" -ea stop
Get-Item : Выполнение команды остановлено, так как переменной оболочки
"ErrorActionPreference" присвоено значение Stop:
Не удается найти путь "C:\несуществующий файл", так как он
не существует.
В строка:1 знак:9
+ Get-Item  <<<< "несуществующий файл" -ea stop
```

Обработка "критических" ошибок (исключений)

Напомним, что "критическими" ошибками в PowerShell называются ошибки, приводящие к прекращению выполнения команды. В традиционных языках программирования такие ошибки обычно называют *исключениями* (exceptions). В PowerShell реализована инструкция Trap, позволяющая перехватывать "критические" ошибки, то есть выполнять определенные действия при их возникновении. Возможны два варианта инструкции Trap:

```
Trap [тип_исключения] {блок_кода}
```

или

```
Trap {блок_кода}
```

В первом случае блок кода будет выполнен только в случае возникновения исключения определенного типа (например, для перехвата ошибок, возникающих при попытках деления на ноль, нужно в качестве типа исключения указать DivideByZeroException), а во втором — при возникновении любого исключения. Приведем пример.

```
PS C:\> Trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
Attempted to divide by zero.
В строка:1 знак:42
+ Trap {"Исключение перехвачено!"} $n=0; 1/$ <<<< n; "n=$n"
n=0
```

В данном случае при вычислении выражения 1/$n возникает исключение (попытка деления на ноль), и управление передается в тело инструкции Trap, где выводится строка "Исключение перехвачено!". Убедимся, что объект, соответствующий ошибке, записывается в массив $Error (поэтому на экран также выводится стандартное сообщение об ошибке):

```
PS C:\> $Error[0]
Attempted to divide by zero.
```

```
В строка:1 знак:48
+ Trap [123] {"Исключение перехвачено!"} $n=0; 1/$ <<<< n; "n=$n"
PS C:\>
```

Обратите внимание, что после выполнения тела инструкции `Trap` управление в нашем примере передалось на команду, идущую после вызвавшей ошибку. На самом деле так происходит не всегда. Если выход из обработчика исключения производится с помощью инструкции `Break`, то оставшиеся команды выполнены не будут:

```
PS C:\> Trap {"Исключение перехвачено!"; Break} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
Attempted to divide by zero.
В строка:1 знак:49
+ Trap {"Исключение перехвачено!"; break} $n=0; 1/$ <<<< n; "n=$n"
PS C:\>
```

Как видите, строка "n=0" на экран не выводится. Если же выход из обработчика исключения производится с помощью инструкции `Continue`, то оставшиеся команды выполняются:

```
PS C:\> Trap {"Исключение перехвачено!"; $_.Exception ; Continue} $n=0;
1/$n; "n=$n"
Исключение перехвачено!
Attempted to divide by zero.
n=0
PS C:\>
```

Данный пример показывает также, что при обработке исключения доступ к соответствующему ему объекту осуществляется, как обычно, с помощью переменной `$_`.

Если выход из обработчика исключения выполнен обычным образом (без применения инструкций `Break` или `Continue`), то дальнейшее поведение определяется значением переменной `$ErrorActionPreference`, рассмотренной в предыдущем разделе. Например, в следующем примере выполнение команд после обработки исключения прерывается:

```
PS C:\> $ErrorActionPreference = "stop"
PS C:\> Trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
Attempted to divide by zero.
В строка:1 знак:42
+ Trap {"Исключение перехвачено!"} $n=0; 1/$ <<<< n; "n=$n"
PS C:\>
```

Изменим теперь режим обработки ошибок и еще раз выполним те же команды после инструкции Trap:

```
PS C:\> $ErrorActionPreference = "silentlycontinue"
PS C:\> Trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
n=0
PS C:\>
```

Как видите, в данном случае на экран не выводится стандартное сообщение об ошибке, и после обработки исключения выполняется следующая команда.

Отладка сценариев

В PowerShell для отладки сценариев предусмотрены несколько механизмов. Мы кратко рассмотрим лишь некоторые из них: вывод на консоль диагностических сообщений, трассировка выполняемых команд и установка точек прерывания.

Вывод диагностических сообщений

Самый простой метод отладки — это вывод на экран сообщений о ходе выполнения сценария. С помощью таких сообщений можно проверить значения переменных или убедиться, что определенная строка сценария выполнена. При этом желательно, чтобы диагностические сообщения выделялись цветом и не попадали в выходной поток, куда записываются все объекты, генерируемые командами PowerShell, а только отображались на экране (консоли). Такую функциональность предоставляет командлет Write-Host:

```
PS C:\> Write-Host "Тестовое сообщение"
Тестовое сообщение
```

Как видите, на экран выводится строка текста. Проверим, что при этом не генерируется никакой объект. Для этого присвоим переменной $a результат выполнения командлета Write-Host и убедимся, что переменная останется неопределенной (значение равно $Null):

```
PS C:\> $a=(Write-Host "Тестовое сообщение")
Тестовое сообщение
PS C:\> $a -eq $Null
True
```

Командлет Write-Host имеет параметры -ForegroundColor и -BackgroundColor, позволяющие задать соответственно цвет текста и цвет фона. Например, следующая команда выведет желтое сообщение на черном фоне:

```
PS C:\> Write-Host "Тест" -ForegroundColor Yellow -BackgroundColor Black
Тест
```

Если после вывода сообщения нужно приостановить выполнение сценария до нажатия клавиши <Enter>, то можно воспользоваться командлетом Read-Host:

```
PS C:\> Read-Host "Нажмите Enter"
Нажмите Enter:

PS C:\>
```

Можно также приостановить выполнение сценария до нажатия произвольной клавиши с помощью следующей команды:

```
PS C:\> $Host.UI.RawUI.ReadKey()

 VirtualKeyCode   Character   ControlKeyState   KeyDown
 --------------   ---------   ---------------   -------
             32                     NumLockOn      True
```

Как видите, в этом случае информация о нажатой на клавиатуре клавише выводится на экран. Для подавления этой информации можно перенаправить результат выполнения метода ReadKey на пустое устройство:

```
PS C:\> $Host.UI.RawUI.ReadKey() > $Null
PS C:\>
```

Командлет Set-PSDebug

В PowerShell основным встроенным инструментом для отладки сценариев является командлет Set-PSDebug. Параметры этого командлета позволяют включить режимы трассировки и пошагового выполнения команд, а также режим обязательного объявления переменных (табл. 9.3).

Таблица 9.3. *Параметры командлета Set-PSDebug*

Параметр	Описание
-Trace 0	Отключение трассировки
-Trace 1	Включение основного режима трассировки
-Trace 2	Включение полного режима трассировки
-Step	Включение режима пошагового выполнения
-Strict	Включение режима обязательного объявления переменных
-Off	Отключение всех механизмов отладки

Рассмотрим режимы отладки более подробно.

Трассировка выполнения команд

Основной режим трассировки включается следующим образом:

```
PS C:\> Set-PSDebug -Trace 1
```

В данном режиме каждая команда, выполняемая интерпретатором, будет отображаться на экране. Например:

```
PS C:\> 3+2
ОТЛАДКА:     1+ 3+2
5
PS C:\> $s="abc"
ОТЛАДКА:     1+ $s="abc"
PS C:\> pwd
ОТЛАДКА:     1+ pwd

Path
----
C:\
```

Отладочная информация выводится другим цветом и предваряется префиксом "ОТЛАДКА". Определим теперь функцию `MyFunc`:

```
PS C:\> Function MyFunc {
>> $x = $Args[0]
>> "x=$x"
>> }
>>
ОТЛАДКА:     1+ Function MyFunc {
```

Вызовем данную функцию в цикле:

```
PS C:\> ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:     2+ $x = $Args[0]
ОТЛАДКА:     3+ "x=$x"
}
x=1
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:     2+ $x = $Args[0]
ОТЛАДКА:     3+ "x=$x"
}
```

```
x=2
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:     2+ $x = $Args[0]
ОТЛАДКА:     3+ "x=$x"
}
x=3
```

В данном случае на экран выводятся все выполняемые команды, однако вызовы функций или присвоение значений переменным явно не выделяются. Включим теперь полный режим трассировки и вновь вызовем функцию `MyFunc` в цикле:

```
PS C:\> Set-PSDebug -Trace 2
ОТЛАДКА:     1+ Set-PSDebug -Trace 2
PS C:\> ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:      ! CALL function 'MyFunc'
ОТЛАДКА:     2+ $x = $Args[0]
ОТЛАДКА:      ! SET $x = '1'.
ОТЛАДКА:     3+ "x=$x"
}
x=1
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:      ! CALL function 'MyFunc'
ОТЛАДКА:     2+ $x = $Args[0]
ОТЛАДКА:      ! SET $x = '2'.
ОТЛАДКА:     3+ "x=$x"
}
x=2
ОТЛАДКА:     1+ ForEach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:      ! CALL function 'MyFunc'
ОТЛАДКА:     2+ $x = $Args[0]
ОТЛАДКА:      ! SET $x = '3'.
ОТЛАДКА:     3+ "x=$x"
}
x=3
```

Как видите, в полном режиме трассировки отдельно выделяются сообщения о вызове функций (префикс `CALL`) и о фактическом присвоении значений переменным (префикс `SET`).

Пошаговое выполнение команд

Рассмотрим еще один режим отладки, позволяющий запускать сценарии в пошаговом режиме (с подтверждением выполнения каждой строки). Включается режим пошагового выполнения команд следующим образом:

```
PS C:\> Set-PSDebug -Step
ОТЛАДКА:    1+ Set-PSDebug -Step
```

Вызовем в цикле функцию `MyFunc`, определенную в предыдущем разделе:

```
PS C:\> ForEach ($i in 1..3) {MyFunc $i}

Продолжить выполнение текущей операции?
   1+ ForEach ($i in 1..3) {MyFunc $i}
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):A
ОТЛАДКА:    1+ ForEach ($i in 1..3) {MyFunc $i}

Продолжить выполнение текущей операции?
   1+ ForEach ($i in 1..3) {MyFunc $i}
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):A
```

В данном режиме на экран выводится текущая строка для исполнения, после чего система запрашивает действие, которое необходимо выполнить. Обратите внимание, что в пошаговом режиме, в отличие от режима трассировки, подробная отладочная информация о выполняемых командах не выводится.

Вложенная командная строка и точки прерывания

В PowerShell реализована возможность рекурсивного вызова интерпретатора во время интерактивного сеанса работы в оболочке или при выполнении сценария. При этом запускается новый (назовем его вложенным) сеанс оболочки в том же командном окне, а текущий сеанс приостанавливается.

Другими словами, можно приостановить выполнение команд в текущем сеансе или сценарии и общаться с PowerShell, используя так называемую *вложенную командную строку*. Это позволяет выполнять команды PowerShell, анализирующие или изменяющие состояние приостановленного сеанса (не нужны отдельные специальные команды для отладки).

Создать новый вложенный сеанс можно в режиме пошагового выполнения команд, выбирая пункт "Приостановить". Включим данный режим:

```
PS C:\> Set-PSDebug -Step
```

Запустим цикл, который будет выводить числа от 1 до 10:

```
PS C:\> $i=0; while ($i++ -lt 10) { $i }

Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) { $i }
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):
```

Нажимая <Enter>, продолжим выполнение шагов до тех пор, пока не будет в первый раз выведено значение переменной `$i`:

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) { $i }
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):
ОТЛАДКА:    1+ $i=0; while ($i++ -lt 10) { $i }

Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) { $i }
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):
ОТЛАДКА:    1+ $i=0; while ($i++ -lt 10) { $i }
1
Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) { $i }
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):
```

На этом шаге мы приостановим выполнение команды, нажав вместо <Enter> клавишу <T>:

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) { $i }
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):T
PS C:\>>>
```

Итак, мы попали во вложенную командную строку (обратите внимание, что изменился вид приглашения). Другим индикатором работы во вложенном сеансе является ненулевое значение переменной `$NestedPromptLevel` (уровень вложения командной строки):

```
PS C:\>>> $NestedPromptLevel
1
```

Во вложенном сеансе можно выполнять те же команды, что и в обычном сеансе PowerShell. Например, проверим, чему равно значение счетчика цикла `$i`:

```
PS C:\>>> $i
2
```

На самом деле можно не только проверять значения переменных, но и изменять их. Например, запишем в переменную `$i` число 8:

```
PS C:\>>> $i=8
```

Выйдем теперь из вложенной командной строки с помощью инструкции `Exit`. На экран вновь будет выдано предложение продолжить выполнение текущей операции:

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) { $i }
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):
```

Запустим все команды, нажав клавишу <X>:

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) { $i }
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех  [T] Приостановить
[?] Справка
(значением по умолчанию является "A"):X
ОТЛАДКА:    1+ $i=0; while ($i++ -lt 10) { $i }
8
9
10
PS C:\>
```

Как видите, переменная `$i` принимает лишь значения 8, 9 и 10, то есть изменения, произведенные во вложенном сеансе, повлияли на приостановленный сеанс.

Итак, режим пошагового выполнения команд позволяет в любой момент вызвать вложенную командную строку для анализа или изменения состояния

интерпретатора. Однако отлаживать более-менее большие сценарии с помощью данного метода часто оказывается неудобно, так как для запуска вложенного сеанса в определенной строке сценария придется каждый раз добираться до этой строки с самого начала сценария (каждую команду при этом нужно выполнять, нажимая клавишу <A>).

Гораздо удобнее было бы использовать *точки прерывания*, позволяющие запускать сценарий в автоматическом режиме и приостанавливать выполнение на нужной команде. В PowerShell аналогом установки точки прерывания можно считать вызов метода `$Host.EnterNestedPrompt()`, например:

```
PS C:\> for ($i=0; $i -lt 10; $i++) {
>>   "i = $i"
>>   if ($i -eq 5) {
>>     "Точка прерывания"
>>     $Host.EnterNestedPrompt()
>>   }
>> }
>>
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
Точка прерывания
PS C:\>>>
```

Как видите, в данном примере на пятой итерации цикла выводится строка "Точка прерывания" и вызывается метод `$Host.EnterNestedPrompt()`. В результате выполнение цикла приостанавливается, и мы попадаем во вложенную командную строку (вид приглашения изменяется). Проверим значение переменной `$i` и присвоим ей новое значение:

```
PS C:\>>> $i
5
PS C:\>>> $i=8
```

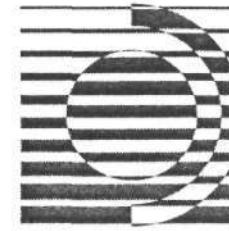
Выйдем из вложенного сеанса с помощью инструкции `Exit`:

```
PS C:\>>> Exit
i = 9
PS C:\>
```

Как видите, приостановленный цикл продолжает выполняться с новым значением переменной-счетчика.

ЧАСТЬ II

Используем PowerShell



Глава 10

Доступ из PowerShell к внешним объектам (COM, WMI, .NET и ADSI)

Одной из основных задач, для решения которых создавалась оболочка PowerShell, было получение из командной строки доступа к различным объектным инфраструктурам, поддерживаемым операционной системой Windows. Раньше к подобным объектам можно было обращаться либо из полноценных приложений с помощью интерфейса прикладного программирования (API), либо из сценариев WSH. В любом случае для использования внешних объектов приходилось писать программный код и изучать их структуру, что значительно затрудняло работу с ними и препятствовало широкому распространению технологий автоматизации среди системных администраторов и пользователей Windows.

Для решения этой проблемы в PowerShell были разработаны специальные командлеты, позволяющие в интерактивном режиме из оболочки обращаться к объектам WMI, COM и .NET.

Работа с COM-объектами

Прежде чем приступить к работе с COM-объектами в оболочке PowerShell, давайте вспомним, как появились эти объекты в операционной системе и для чего они нужны.

В Windows с самого начала развития этой ОС для обеспечения обмена данными между приложениями была разработана технология *связывания и внедрения объектов* (Object Linking and Embedding, OLE). Вначале технология OLE использовалась для создания составных документов, а затем для решения более общей задачи — предоставления приложениями друг другу собственных функций (служб) и правильного использования этих функций. Технология, позволяющая одному приложению (*клиенту автоматизации*) вызывать функции другого приложения (*сервера автоматизации*) была

названа OLE Automation. В основе OLE и OLE Automation лежала разработанная Microsoft базовая технология Component Object Model (COM), которая предоставляла двоичный (не зависящий от языка программирования) стандарт для программных компонентов. Объекты, созданные по этому стандарту и зарегистрированные определенным образом в операционной системе, могли использоваться в других приложениях без применения языков, средств или систем программирования, а только с помощью двоичных компонентов (исполняемых файлов или динамических библиотек).

С середины 90-х годов прошлого столетия вместо термина OLE стал использоваться новый термин — *ActiveX*, первоначально обозначавший WWW-компоненты (объекты), созданные на базе технологии COM.

До появления платформы .NET технология ActiveX была ключевой в продуктах Microsoft. В настоящее время COM-объекты по-прежнему широко используются в Windows. Многие приложения как компании Microsoft (Microsoft Office, Internet Explorer, Internet Information Service (IIS) и т. д.), так и сторонних разработчиков являются серверами автоматизации, предоставляя через COM-объекты доступ к своим службам. Другими словами, сервера автоматизации позволяют управлять некоторыми аспектами соответствующего приложения внешним программам.

> ***Замечание***
>
> Некоторые COM-объекты, которые могут оказаться полезными при автоматизации задач администратора операционной системы, описаны в *приложении 2*.

Работая в PowerShell, мы будем идентифицировать COM-объекты по их *программным идентификаторам* (ProgID) — символьным псевдонимам, назначаемым при регистрации объектов в системе. Согласно общепринятому соглашению идентификаторы ProgID имеют следующий вид (при этом общая длина программного идентификатора не должна превышать 39 символов):

```
Библиотека_типов.Класс.Версия
```

или просто

```
Библиотека_типов.Класс
```

Перед первой точкой в ProgID стоит имя *библиотеки типов* (type library) для объекта, которая может существовать как в виде отдельного файла с расширением tlb, так и в виде части файла с исполняемым кодом объекта. Библиотека типов, содержащая сведения о COM-объекте, регистрируется в системном реестре при установке приложения, публикующего этот объект. Довольно часто имя библиотеки типов совпадает с именем приложения, являющегося сервером COM-объектов. После точки в ProgID указывается имя класса, содержащего свойства и методы COM-объекта, доступные для использования

другими приложениями. Номер версии при создании экземпляров COM-объектов, как правило, не используется. Вот несколько примеров ProgID: `InternetExplorer.Application` (приложение Internet Explorer), `Word.Application` (приложение Microsoft Word), `WScript.Shell` (класс `Shell` из объектной модели сервера сценариев Windows Script Host).

В PowerShell имеется командлет `New-Object`, позволяющий, в частности, создавать экземпляры внешних COM-объектов, указывая соответствующий ProgID в качестве значения параметра `ComObject`. Например, экземпляр COM-объекта с программным идентификатором `WScript.Shell` создается следующим образом:

```
PS C:\> $Shell = New-Object -ComObject WScript.Shell
```

Выполняя командлет `New-Object`, интерпретатор PowerShell через ProgID получает из системного реестра путь к файлам нужной библиотеки типов. Затем с помощью этой библиотеки в память загружается экземпляр запрашиваемого объекта, и его интерфейсы становятся доступными для использования в PowerShell. Ссылка на созданный объект сохраняется в переменной; в дальнейшем, используя эту переменную, мы получаем доступ к свойствам и методам объекта, а также к его вложенным объектам (если они имеются).

> ***Замечание***
>
> "Общение" с COM-объектами в PowerShell происходит с помощью соответствующих механизмов .NET Framework (создаются экземпляры .NET-класса `System.__ComObject`). Поэтому на командлет `New-Object` действуют те же ограничения, какие применяются в платформе .NET во время вызова COM-объектов.

Посмотрим, какие свойства и методы имеются у COM-объекта `WScript.Shell`. Для этого воспользуемся, как обычно, командлетом `Get-Member`, передав ему по конвейеру переменную `$Shell`, в которой сохранена ссылка на данный COM-объект:

```
PS C:\> $Shell | Get-Member

   TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd350...

Name                       MemberType          Definition
----                       ----------          ----------
AppActivate                Method              bool AppActivate (...
CreateShortcut             Method              IDispatch CreateSh...
Exec                       Method              IWshExec Exec (str...
ExpandEnvironmentStrings   Method              string ExpandEnvir...
```

```
LogEvent                Method                  bool LogEvent (Var...
Popup                   Method                  int Popup (string,...
RegDelete               Method                  void RegDelete (st...
RegRead                 Method                  Variant RegRead (s...
RegWrite                Method                  void RegWrite (str...
Run                     Method                  int Run (string, V...
SendKeys                Method                  void SendKeys (str...
Environment             ParameterizedProperty   IWshEnvironment En...
CurrentDirectory        Property                string CurrentDire...
SpecialFolders          Property                IWshCollection Spe...
```

Попробуем воспользоваться каким-нибудь методом COM-объекта `WScript.Shell`. Например, метод `CreateShortcut` позволяет быстро создавать ярлыки для папок и файлов. Для примера мы создадим на рабочем столе активного пользователя ярлык PSHome.lnk для папки, в которой установлена оболочка PowerShell.

Путь к рабочему столу можно определить разными способами, например, с помощью всегда определенной в PowerShell переменной `$Home`, в которой хранится путь к личной папке активного пользователя:

```
PS C:\> $Home
C:\Documents and Settings\User
```

По умолчанию содержимое рабочего стола хранится в подкаталоге "Рабочий стол" личной папки пользователя, поэтому для создания ярлыка выполним следующую команду (напомним, что внутри строки в двойных кавычках имя переменной заменяется ее значением):

```
PS C:\> $lnk = $Shell.CreateShortcut("$Home\Рабочий стол\PSHome.lnk")
```

> ***Замечание***
>
> Путь к рабочему столу и другим специальным папкам активного пользователя можно также получить с помощью COM-объекта `WScript.Shell` (см. *приложение 2*). Более того, при написании сценариев правильнее именно так и поступать, а не вписывать строку в сценарий, поскольку тогда программа будет работать корректно и в других языковых версиях, а также, вероятнее всего, и в новых версиях операционной системы. А при работе в командной строке проще написать название папки в виде строки, как и сделано выше.

Посмотрим, какие свойства и методы имеет объект, сохраненный в переменной `$lnk`:

```
PS C:\> $lnk | Get-Member

   TypeName: System.__ComObject#{f935dc23-1cf0-11d0-adb9-00c04fd58a0b}
```

```
Name             MemberType Definition
----             ---------- ----------
Load             Method     void Load (string)
Save             Method     void Save ()
Arguments        Property   string Arguments () {get} {set}
Description      Property   string Description () {get} {set}
FullName         Property   string FullName () {get}
Hotkey           Property   string Hotkey () {get} {set}
IconLocation     Property   string IconLocation () {get} {set}
RelativePath     Property    {get} {set}
TargetPath       Property   string TargetPath () {get} {set}
WindowStyle      Property   int WindowStyle () {get} {set}
WorkingDirectory Property   string WorkingDirectory () {get} {set}
```

Для создания ярлыка для папки достаточно присвоить свойству `TargetPath` путь к ней и вызвать метод `Save` для сохранения ярлыка. Путь к домашней папке PowerShell хранится в специальной переменной `$PSHome`, поэтому выполняем следующие команды:

```
PS C:\> $lnk.TargetPath = $PSHome
PS C:\> $lnk.Save()
```

Задача решена, ярлык на рабочем столе активного пользователя создан. Полный текст сценария для создания ярлыка приведен в листинге 10.1.

Листинг 10.1. Создание ярлыка на рабочем столе (файл CreateShortcut.ps1)

```
#################################################################
# Имя: CreateShortcut.ps1
# Язык: PoSH 1.0
# Описание: Создание на рабочем столе активного пользователя
#           ярлыка для домашней папки PowerShell
#################################################################

#Создаем экземпляр COM-объекта
$Shell = New-Object -ComObject WScript.Shell

#Создаем ярлык
$lnk = $Shell.CreateShortcut("$Home\Рабочий стол\PSHome.lnk")
```

```
#Заполняем свойства ярлыка
$lnk.TargetPath = $PSHome

#Сохраняем ярлык
$lnk.Save()
### Конец сценария ########################
```

Внешние серверы автоматизации на примере Microsoft Office

Одними из самых распространенных и часто используемых серверов автоматизации в Windows являются приложения пакета Microsoft Office. Мы рассмотрим на примерах, каким образом можно выводить из PowerShell информацию в две наиболее распространенные программы этого пакета: Microsoft Word и Microsoft Excel. Однако предварительно скажем несколько слов об объектных моделях, представляемых данными программами.

Объектные модели Microsoft Word и Excel

Для того чтобы использовать из сценариев PowerShell те возможности, которые поддерживают программы Word и Excel, необходимо знать, какие именно объекты предоставляются для внешнего использования этими серверами автоматизации и как эти объекты соотносятся друг с другом. Хотя объектные модели приложений Microsoft Office довольно сложны, они похожи друг на друга, причем для практических целей достаточно понять принцип работы с несколькими ключевыми объектами. Здесь мы не будем подробно рассматривать свойства и методы объектов Word и Excel, а лишь кратко упомянем, какие именно объекты будут использоваться в рассмотренных далее примерах сценариев.

На самом верхнем уровне объектной модели Word находится объект `Application`, который представляет непосредственно само приложение Word и содержит (в качестве свойств) все остальные объекты. Таким образом, объект `Application` используется для получения доступа к любому другому объекту Word.

Семейство `Documents` является свойством объекта `Application` и содержит набор объектов `Document`, каждый из которых соответствует открытому в Word документу. Класс `Documents` понадобится нам в сценариях для создания новых документов. Объект `Document` содержит в качестве своих свойств семейства различных объектов документа: символов (`Characters`), слов (`Words`), предложений (`Sentences`), параграфов (`Paragraphs`), закладок (`Bookmarks`) и т. д.

Объект `Selection` позволяет работать с выделенным фрагментом текста (этот фрагмент может быть и пустым). Таким образом, можно сказать, что объект `Selection` открывает путь в документ, так как он предоставляет доступ к выделенному фрагменту документа. Используя свойства этого объекта (которые, в свою очередь, могут являться объектами со своими свойствами), можно управлять параметрами выделенного фрагмента, например, устанавливать нужный размер и гарнитуру шрифта, выравнивать параграфы по центру и т. п. Также объект `Selection` имеет ряд полезных методов. В частности, у него имеется метод `TypeText(Text)`, с помощью которого можно вставлять текст в документ.

Объектная модель Excel построена по тому же принципу, что и объектная модель Word. Основным объектом, содержащим все остальные, является `Application`. Напомним, что отдельные файлы в Excel называются *рабочими книгами*. Семейство `Workbooks` в Excel является аналогом семейства `Documents` в Word и содержит набор объектов `Workbook` (аналог объекта `Document` в Word), каждый из которых соответствует открытой в Word рабочей книге. Новая рабочая книга создается с помощью метода `Add()` объекта `Workbooks`.

Для доступа к ячейкам активного рабочего листа Excel используется свойство `Cells` объекта `Application`. В общем случае это свойство возвращает объект `Range`, представляющий набор ячеек. Для получения или изменения значения отдельной ячейки применяется конструкция `Cells.Item(row, column).Value`, где *`row`* и *`column`* являются соответственно номерами строки и столбца (начиная с единицы), на пересечении которых находится данная ячейка.

В Excel, как и в Word, имеется объект `Selection`, позволяющий работать с выделенным фрагментом электронной таблицы. Самым простым способом выделить диапазон ячеек активного рабочего листа является использование метода `Select()` объекта `Range`. Например, выражение `Range("A1:C1").Select()` позволяет выделить три смежные ячейки: `"A1"`, `"B1"` и `"C1"`.

Для того чтобы понять, какой именно объект Word или Excel нужно использовать для решения той или иной задачи, часто проще всего бывает проделать в соответствующем приложении необходимые манипуляции вручную, включив предварительно режим записи макроса. В результате мы получим текст макроса на языке VBA (Visual Basic for Applications), из которого будет ясно, какие методы и с какими параметрами нужно вызывать, а также какие значения нужно присваивать свойствам объектов. В качестве простой иллюстрации проделаем следующие действия. Запустим Word, запустим Macro Recorder (**Сервис | Макрос | Начать запись...**), назовем новый макрос "Пример1" и нажмем кнопку **OK**. После этого напишем в документе слово

"Пример" и прекратим запись макроса. Теперь можно посмотреть содержимое записанного макроса. Для этого нужно выбрать пункт **Макросы...** в меню **Сервис | Макрос**, выделить макрос "Пример1" в списке всех доступных макросов и нажать кнопку **Изменить**. В открывшемся окне редактора Visual Basic появится текст макроса:

```
Sub Пример1()
'
' Пример1 Макрос
' Макрос записан 12.03.08 Андрей Владимирович Попов
'
    Selection.TypeText Text:="Пример"
End Sub
```

Как мы видим, для печати слова в документе был использован метод `TypeText` объекта `Selection`.

Макросы в Excel записываются и редактируются аналогичным образом.

Взаимодействие с Microsoft Word

Давайте научимся управлять с помощью PowerShell программой Microsoft Word: запустим из оболочки данную программу и напечатаем в окне Word строку текста.

Сделать это несложно. Сначала создаем экземпляр главного объекта сервера автоматизации Microsoft Word, который имеет программный идентификатор `Word.Application`, и сохраняем ссылку на этот объект в переменной `$Word`:

```
PS C:\> $Word = New-Object -ComObject Word.Application
```

С помощью метода `Add` коллекции `Documents` создаем новый документ и сохраняем ссылку на него в переменной `$doc`:

```
PS C:\> $doc = $Word.Documents.Add()
```

Делаем окно с новым документом видимым. Для этого нужно в свойство `Visible` объекта `$Word` записать значение `$True`:

```
PS C:\> $Word.Visible = $True
```

В переменную `$sel` занесем ссылку на объект `Selection`, с помощью которого можно задавать тип и размер шрифта, тип выравнивания абзацев, а также печатать в документе строки текста:

```
PS C:\> $sel = $Word.Selection
```

Установим размер шрифта:

```
PS C:\> $sel.Font.Size = 14
```

Установим полужирный шрифт:

```
PS C:\> $sel.Font.Bold = $True
```

Наконец, печатаем строку текста:

```
PS C:\> $sel.TypeText("Привет из PowerShell!" )
```

Полный текст сценария для управления приложением Microsoft Word приведен в листинге 10.2.

Листинг 10.2. Управление приложением Microsoft Word (файл PrintInWord.ps1)

```
#################################################################
# Имя: PrintInWord.ps1
# Язык: PoSH 1.0
# Описание: Управление приложением Microsoft Word
#################################################################
#Создаем экземпляр сервера автоматизации
$Word = New-Object -ComObject Word.Application

#Создаем новый документ
$doc = $Word.Documents.Add()

#Делаем видимым окно приложения
$Word.Visible = $True

#Создаем объект Selection
$sel = $Word.Selection

#Устанавливаем размер шрифта
$sel.Font.Size=14

#Устанавливаем полужирный шрифт
$sel.Font.Bold=$True

#Печатаем строку текста в Word
$sel.TypeText("Привет из PowerShell!" )
### Конец сценария ########################
```

Взаимодействие с Microsoft Excel

Давайте теперь создадим из оболочки PowerShell новую рабочую книгу Microsoft Excel и выведем в нее заголовок и одну запись телефонной книги (рис. 10.1).

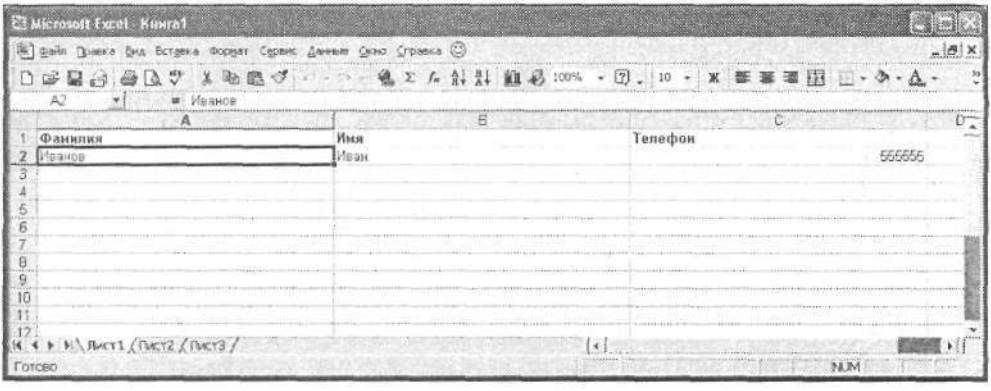


Рис. 10.1. Рабочая книга Microsoft Excel, созданная из оболочки PowerShell

Как и в случае с Microsoft Word, сначала мы создаем экземпляр главного объекта `Excel.Application` и сохраняем ссылку на этот экземпляр в переменной `$XL`:

```
PS C:\> $XL = New-Object -ComObject Excel.Application
```

В результате запускается программа Microsoft Excel. Для того чтобы сделать окно приложения видимым, установим значение свойства `Visible` в `$True`:

```
PS C:\> $XL.Visible = $True
```

Создать новую рабочую книгу можно с помощью метода `Add` коллекции `WorkBooks`; при выполнении данного метода на экран выводится много ненужной технической информации, которую мы перенаправим на пустое устройство `$Null`:

```
PS C:\> $XL.WorkBooks.Add() > $Null
```

Теперь можно устанавливать нужную ширину столбцов (свойство `ColumnWidth`) и выводить текст заголовка:

```
PS C:\> $XL.Columns.Item(1).ColumnWidth = 40
PS C:\> $XL.Columns.Item(2).ColumnWidth = 40
PS C:\> $XL.Columns.Item(3).ColumnWidth = 40
PS C:\> $XL.Cells.Item(1,1).Value="Фамилия"
PS C:\> $XL.Cells.Item(1,2).Value="Имя"
PS C:\> $XL.Cells.Item(1,3).Value="Телефон"
```

С помощью метода `Select()` выделим три ячейки с введенным заголовком и установим в них полужирный шрифт:

```
PS C:\> $XL.Range("A1:C1").Select() > $Null
PS C:\> $XL.Selection.Font.Bold = $true
```

Во вторую строку рабочей книги выводим данные об абоненте:

```
PS C:\> $XL.Cells.Item(2,1).Value="Иванов"
PS C:\> $XL.Cells.Item(2,2).Value="Иван"
PS C:\> $XL.Cells.Item(2,3).Value="555555"
```

Полный текст сценария для управления приложением Microsoft Excel приведен в листинге 10.3.

Листинг 10.3. Управление приложением Microsoft Excel (файл PrintInExcel.ps1)

```
################################################################
# Имя: PrintInExcel.ps1
# Язык: PoSH 1.0
# Описание: Управление приложением Microsoft Excel
################################################################

#Создаем объект-приложение Microsoft Excel
$XL = New-Object -ComObject Excel.Application

#Делаем окно Microsoft Excel видимым
$XL.Visible = $True

#Открываем новую рабочую книгу
$XL.WorkBooks.Add() > $Null

#Устанавливаем нужную ширину колонок
$XL.Columns.Item(1).ColumnWidth = 40
$XL.Columns.Item(2).ColumnWidth = 40
$XL.Columns.Item(3).ColumnWidth = 40

#Печатаем в ячейках текст
$XL.Cells.Item(1,1).Value="Фамилия"
$XL.Cells.Item(1,2).Value="Имя"
$XL.Cells.Item(1,3).Value="Телефон"
```

```
#Выделяем три ячейки
$XL.Range("A1:C1").Select() > $Null

#Устанавливаем полужирный текст для выделенного диапазона
$XL.Selection.Font.Bold = $true

#Печатаем в ячейках текст
$XL.Cells.Item(2,1).Value="Иванов"
$XL.Cells.Item(2,2).Value="Иван"
$XL.Cells.Item(2,3).Value="555555"
### Конец сценария ########################
```

Доступ к объектам WMI

Перейдем к работе с еще одной, занимающей особое место среди инструментов и средств автоматизации в операционной системе Windows, объектной моделью — WMI (Windows Management Instrumentation).

Технология WMI — это созданная фирмой Microsoft реализация модели управления предприятием на базе Web (Web-Based Enterprise Management, WBEM). Эта технология разработана и принята рабочей группой по управлению распределенными системами (Distributed Management Task Force, DMTF) при участии таких компаний, как BMC Software, Cisco Systems, Intel и Microsoft. Задачей DMTF была разработка таких стандартов для удаленного управления информационной средой предприятия, которые позволили бы управлять всеми физическими и логическими компонентами этой среды из одной точки и не зависели бы при этом от конкретного оборудования, сетевой инфраструктуры, операционной системы, файловой системы и т. д.

Для этого была предложена схема CIM (Common Information Model), которая представляет физическую и логическую структуры компьютерной системы в виде единой расширяемой объектно-ориентированной информационной модели и определяет единые интерфейсы для получения информации о любом компоненте этой модели.

> ***Замечание***
>
> Более подробное описание структуры CIM и принципов взаимодействия компонентов WMI приведено в *приложении 1*.

Несмотря на то, что внутри операционной системы Windows технология WMI используется уже давно (начиная с версии Windows 98), большинство пользователей и многие администраторы до сих пор не знакомы с ней. Это связано с тем, что до появления PowerShell не было простого и удобного

стандартного инструмента для работы с WMI. Для обращения к нужным объектам нужно было либо писать программу на C++, либо создавать сценарий на языке VBScript или JScript, либо изучать специальную оболочку WMIC (WMI Command Line) со своим специфическим языком. Любой из этих подходов был слишком сложен для большинства администраторов и пользователей, не имеющих навыков в программировании.

Оболочка PowerShell превращает WMI в удобный инструмент командной строки, доступный рядовому пользователю.

Подключение к подсистеме WMI. Получение списка классов

В PowerShell экземпляры объектов WMI можно получать с помощью специального командлета `Get-WmiObject` (псевдоним `gwmi`). При этом для обращения к определенному объекту WMI нужно знать наименование класса, к которому он относится (например, `Win32_Process` для процессов, запущенных в системе, или `Win32_Service` для зарегистрированных на компьютере служб).

> ***Замечание***
>
> Для первоначального знакомства с объектной моделью WMI хорошо подходят утилиты из разработанного компанией Microsoft пакета WMI Tools, позволяющих просматривать иерархическую структуру классов и связи между различными объектами CIM. Краткое описание пакета WMI Tools приведено в *приложении 1*.

PowerShell позволяет в интерактивном режиме получить список всех классов WMI на локальном или удаленном компьютере. Для этого нужно выполнить командлет `Get-WmiObject` с параметром `-List`:

```
PS C:\> Get-WmiObject -List

__SecurityRelatedClass                __NTLMUser9X
__PARAMETERS                          __SystemSecurity
__NotifyStatus                        __ExtendedStatus
Win32_PrivilegesStatus                Win32_TSNetworkAdapterSettingE...
Win32_TSRemoteControlSettingError     Win32_TSEnvironmentSettingError
Win32_TSSessionDirectoryError         Win32_TSLogonSettingError
Win32_TerminalError                   Win32_JobObjectStatus
Win32_TerminalServiceSettingError     Win32_TSPermissionsSettingError
Win32_TSClientSettingError            Win32_TSGeneralSettingError. . .
```

В данном примере выводится список классов WMI на локальном компьютере. Если вам нужно подключиться к подсистеме WMI на другой машине,

то ее имя или IP-адрес нужно указать в качестве значения параметра -ComputerName, например:

```
PS C:\> Get-WmiObject -ComputerName 10.169.1.15 -List

__SecurityRelatedClass                __NTLMUser9X
__PARAMETERS                          __SystemSecurity
__NotifyStatus                        __ExtendedStatus
Win32_PrivilegesStatus                Win32_TSNetworkAdapterSettingE...
Win32_TSRemoteControlSettingError     Win32_TSEnvironmentSettingError
Win32_TSSessionDirectoryError         Win32_TSLogonSettingError
Win32_TerminalError                   Win32_JobObjectStatus
Win32_TerminalServiceSettingError     Win32_TSPermissionsSettingError
Win32_TSClientSettingError            Win32_TSGeneralSettingError
. . .
PS C:\> Get-WmiObject -ComputerName Comp1 -List

__SecurityRelatedClass                __NTLMUser9X
__PARAMETERS                          __SystemSecurity
__NotifyStatus                        __ExtendedStatus
Win32_PrivilegesStatus                Win32_TSNetworkAdapterSettingE...
Win32_TSRemoteControlSettingError     Win32_TSEnvironmentSettingError
Win32_TSSessionDirectoryError         Win32_TSLogonSettingError
Win32_TerminalError                   Win32_JobObjectStatus
Win32_TerminalServiceSettingError     Win32_TSPermissionsSettingError
Win32_TSClientSettingError            Win32_TSGeneralSettingError
. . .
```

По умолчанию командлет Get-WmiObject подключается к пространству имен Root\CIMV2, хранящему большое количество классов для управления компьютерной системой. Иногда может потребоваться класс или объект из другого пространства имен (например, Root). Сменить подключаемое пространство имен позволяет параметр -Namespace, в качестве значения которого указывается нужное наименование. Например, следующая команда выведет список классов из пространства имен Root:

```
PS C:\> Get-WmiObject -Namespace Root -List

__SecurityRelatedClass                __NTLMUser9X
__PARAMETERS                          __SystemSecurity
```

```
__NotifyStatus                  __ExtendedStatus
__SystemClass                   __Provider
__Win32Provider                 __thisNAMESPACE
__IndicationRelated             __EventGenerator
__TimerInstruction              __IntervalTimerInstruction
__AbsoluteTimerInstruction      __Event
__NamespaceOperationEvent       __NamespaceDeletionEvent
__NamespaceCreationEvent        __NamespaceModificationEvent
__InstanceOperationEvent        __MethodInvocationEvent
__InstanceCreationEvent         __InstanceModificationEvent
. . .
```

Получение объектов WMI

Зная имя класса WMI, получить экземпляры этого класса очень просто. Достаточно просто указать название нужного класса в качестве параметра командлета `Get-WmiObject`. Например, следующая команда выведет все экземпляры класса `Win32_Service`, соответствующие службам, зарегистрированным на локальном компьютере:

```
PS C:\> Get-WmiObject Win32_Service

ExitCode  : 1077
Name      : Alerter
ProcessId : 0
StartMode : Disabled
State     : Stopped
Status    : OK

ExitCode  : 0
Name      : ALG
ProcessId : 564
StartMode : Manual
State     : Running
Status    : OK
. . .
```

Итак, данная команда вывела информацию о службах, зарегистрированных в системе. На самом деле объекты класса `Win32_Service` имеют намного больше свойств, чем по умолчанию отображается на экране; увидеть список

всех свойств и методов объекта WMI можно, как и в случае .NET-объектов, с помощью командлета `Get-Member`:

```
PS C:\> Get-WmiObject Win32_Service | Get-Member

   TypeName: System.Management.ManagementObject#root\cimv2\Win32_Ser...

Name                      MemberType   Definition
----                      ----------   ----------
Change                    Method       System.Management.Management...
ChangeStartMode           Method       System.Management.Management...
Create                    Method       System.Management.Management...
InterrogateService        Method       System.Management.Management...
PauseService              Method       System.Management.Management...
ResumeService             Method       System.Management.Management...
StartService              Method       System.Management.Management...
StopService               Method       System.Management.Management...
UserControlService        Method       System.Management.Management...
AcceptPause               Property     System.Boolean AcceptPause {...
AcceptStop                Property     System.Boolean AcceptStop {g...
Caption                   Property     System.String Caption {get;s...
CheckPoint                Property     System.UInt32 CheckPoint {ge...
CreationClassName         Property     System.String CreationClassN...
Description               Property     System.String Description {g...
DesktopInteract           Property     System.Boolean DesktopIntera...
DisplayName               Property     System.String DisplayName {g...
ErrorControl              Property     System.String ErrorControl {...
ExitCode                  Property     System.UInt32 ExitCode {get;...
InstallDate               Property     System.String InstallDate {g...
Name                      Property     System.String Name {get;set;...
. . .
```

Используя командлеты форматирования, можно выводить на экран интересующие нас свойства. Например, следующая команда выведет имена всех служб (свойство `Name`) и логическое значение (свойство `Started`), показывающее, была ли запущена соответствующая служба:

```
PS C:\> Get-WmiObject Win32_Service | Format-Table Name, Started

Name                                    Started
----                                    -------
Alerter                                   False
ALG                                        True
```

```
AppMgmt                                  False
aspnet_state                             False
Ati HotKey Poller                         True
AudioSrv                                  True
BITS                                      True
Browser                                   True
cisvc                                    False
ClipSrv                                  False
clr_optimization_v2.0.50727_32           False
COMSysApp                                False
CryptSvc                                  True
. . .
```

Как упоминалось в *главе 6*, работа с любыми внешними объектами в PowerShell производится с помощью системы адаптации типов, поэтому к объектам WMI *можно применять любые* командлеты для фильтрации, сортировки, группировки и т. д. Например, для вывода служб, запускаемых автоматически при загрузке системы, можно выполнить следующий конвейер команд:

```
PS C:\> Get-WmiObject Win32_Service |
>> Where-Object {$_.StartMode -eq "Auto"}
>>

ExitCode  : 0
Name      : Ati HotKey Poller
ProcessId : 1616
StartMode : Auto
State     : Running
Status    : OK

ExitCode  : 0
Name      : AudioSrv
ProcessId : 1132
StartMode : Auto
State     : Running
Status    : OK
. . .
```

Здесь с помощью командлета `Where-Object` из всех экземпляров класса `Win32_Service` отбираются объекты, у которых свойство `StartMode` равно "Auto".

Выполнение WQL-запросов

В предыдущем разделе было показано, что для выборки объектов WMI по определенным критериям можно пользоваться конвейерами с командлетами PowerShell. Этот подход универсален, он срабатывает для объектов любых типов, однако для объектов WMI имеется альтернатива: специальный язык запросов WMI Query Language (WQL). Этот язык является частью инфраструктуры WMI, поэтому WQL-запросы можно выполнять в интерактивных утилитах или сценариях на языках VBScript или JScript.

> ***Замечание***
>
> Подробную информацию о возможностях языка WQL можно найти в электронной библиотеке MSDN (Microsoft Developer Network) на сайте компании Microsoft.

В PowerShell запросы на языке WQL можно выполнять с помощью параметра `-Query` командлета `Get-WmiObject`. Например, для выборки всех служб, запускаемых автоматически при запуске системы, можно выполнить следующий WQL-запрос:

```
select * from win32_service where startmode="Auto"
```

В PowerShell данный запрос выполняется следующим образом:

```
PS C:\> Get-WmiObject -Query 'select * from win32_service where start-
mode="Auto"'

ExitCode  : 0
Name      : Ati HotKey Poller
ProcessId : 1616
StartMode : Auto
State     : Running
Status    : OK

ExitCode  : 0
Name      : AudioSrv
ProcessId : 1132
StartMode : Auto
State     : Running
Status    : OK
. . .
```

Использование объектов .NET

Как уже неоднократно упоминалось ранее, оболочка PowerShell базируется на платформе .NET Framework и позволяет использовать все ее возможности из командной строки. В *главе 5* мы уже работали с объектной моделью .NET, вызывая статические методы без генерации экземпляров классов.

Создавать экземпляры .NET-объектов позволяет командлет `New-Object`, которым мы уже пользовались при работе с COM-объектами. К этой возможности можно прибегнуть в случае, если для решения определенной задачи не удается найти подходящий командлет PowerShell, но существует .NET-объект, который обладает нужной функциональностью. Например, большинство командлетов PowerShell 1.0 не поддерживают работу с удаленными компьютерами, в частности, нельзя управлять журналами событий операционной системы на удаленном компьютере. Однако можно в оболочке создать экземпляр .NET-объекта `System.Diagnostics.EventLog`, сопоставив его с журналом событий на определенном компьютере, и воспользоваться методами этого объекта для очистки журнала или настройки параметров протоколирования событий (подробнее данная задача рассматривается в *главе 14*).

Следует иметь в виду, что система классов .NET создавалась в первую очередь для разработчиков сложного программного обеспечения, поэтому сама по себе объектная модель .NET требует отдельного рассмотрения и изучения. Подробное рассмотрение программирования сценариев, работающих с объектами платформы .NET, выходит за рамки настоящей книги. Мы рассмотрим лишь простой пример, показывающий, как можно с помощью PowerShell работать с библиотекой .NET Windows Form (WinForms), которая содержит объекты для построения графических приложений Windows (форм).

Платформа .NET построена таким образом, что для обращения к тем или иным объектам нужно предварительно загрузить в операционную память соответствующую *сборку* (assembly) — динамическую библиотеку определенного вида. Наиболее часто использующиеся сборки загружаются в PowerShell автоматически, их список можно увидеть с помощью статического метода `getAssemblies()` следующим образом:

```
PS C:\> [AppDomain]::CurrentDomain.getAssemblies()

GAC    Version        Location
---    -------        --------
True   v2.0.50727     C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ms...
True   v2.0.50727     C:\WINDOWS\assembly\GAC_MSIL\Microsoft.PowerShel...
True   v2.0.50727     C:\WINDOWS\assembly\GAC_MSIL\System\2.0.0.0__b77...
```

```
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\System.Management.A...
True  v2.0        C:\WINDOWS\assembly\GAC_MSIL\Microsoft.PowerShel...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\System.Configuratio...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\Microsoft.PowerShel...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\Microsoft.PowerShel...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\Microsoft.PowerShel...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_32\System.Data\2.0.0.0__...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\System.Xml\2.0.0.0_...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\System.DirectorySer...
True  v2.0.50727  C:\WINDOWS\assembly\GAC_MSIL\System.Management\2...
True  v2.0        C:\WINDOWS\assembly\GAC_MSIL\System.Management.A...
True  v2.0        C:\WINDOWS\assembly\GAC_MSIL\Microsoft.PowerShel...
True  v2.0        C:\WINDOWS\assembly\GAC_MSIL\Microsoft.PowerShel
```

Для обращения к объектам WinForms нужно с помощью метода `LoadWithPartialName` загрузить сборку, поддерживающие эти объекты (результат выполнения метода приводится к типу `void` для подавления вывода на экран ненужной информации):

```
PS C:\> [void][System.Reflection.Assembly]::LoadWithPartialName(
>> "System.Windows.Forms")
>>
PS C:\>
```

Любое графическое приложение должно иметь главную форму — объект типа `Windows.Forms.Form`. Мы сохраним данную форму в переменной `$form`:

```
PS C:\> $form = New-Object Windows.Forms.Form
```

В заголовок формы (свойство `Text` объекта `$form`) запишем строку "Первая форма":

```
PS C:\> $form.Text = "Первая форма"
```

Теперь создадим кнопку (объект типа `Windows.Forms.Button`) с надписью "Нажми!":

```
PS C:\> $button = New-Object Windows.Forms.Button
PS C:\> $button.Text = "Нажми!"
```

Пусть наша кнопка будет занимать все пространство формы. Для этого в свойство `Dock` объекта `$button` запишем строку "fill":

```
PS C:\> $button.Dock = "fill"
```

Определим теперь действие, которое будет выполняться при нажатии кнопки. Для этого нужно написать обработчик события `Click` кнопки (то есть указать, какие команды должны выполниться при нажатии на кнопку). Обработ-

чик событий — это специальный метод с названием `Add_Событие`. Пусть в нашем случае нажатие кнопки будет приводить к закрытию формы (метод `Close` объекта `$form`):

```
PS C:\> $button.Add_Click({$form.Close()})
```

Теперь поместим кнопку на форму с помощью метода `Add` коллекции `Controls` объекта `$form`:

```
PS C:\> $form.Controls.Add($button)
```

Для того чтобы наша форма стала активной при отображении, нужно определить обработчик события `Shown`, поместив в этот обработчик вызов метода `Activate` объекта `$form`:

```
PS C:\> $form.Add_Shown({$form.Activate()})
```

Теперь осталось вызвать метод `ShowDialog` для вывода нашей формы на экран (рис. 10.2):

```
PS C:\> $form.ShowDialog()
```

Рис. 10.2. Графическая форма, созданная из оболочки PowerShell

Полный текст сценария для создания графической формы приведен в листинге 10.4.

Листинг 10.4. Создание графической формы (файл WinForm.ps1)

```
#################################################################
# Имя: WinForm.ps1
# Язык: PoSH 1.0
# Описание: Создание графической формы из сценария PowerShell
```

```
#################################################################
#Загружаем сборку System.Windows.Forms
[void][System.Reflection.Assembly]::LoadWithPartialName(
    "System.Windows.Forms")
#Создаем главную форму - объект Windows.Forms.Form
$form = New-Object Windows.Forms.Form
#Заполняем заголовок формы
$form.Text = "Первая форма"

#Создаем объект-кнопку
$button = New-Object Windows.Forms.Button
#Задаем текст кнопки
$button.Text = "Нажми!"
#Определяем расположение кнопки
$button.Dock = "fill"

#Определяем обработчик нажатия кнопки
$button.Add_Click({$form.Close()})
#Добавляем кнопку на форму
$form.Controls.Add($button)
#Определяем обработчик события Shown для активизации формы
$form.Add_Shown({$form.Activate()})

#Выводим кнопку на экран
$form.ShowDialog()
### Конец сценария ########################
```

Доступ к службе каталогов ADSI

Обсудим сначала термины "каталог" и "служба каталога", которые будут использоваться в этом разделе. Под *каталогом* в общем смысле этого слова подразумевается источник информации, в котором хранятся данные о некоторых объектах. Например, в телефонном каталоге хранятся сведения об абонентах телефонной сети, в библиотечном каталоге — данные о книгах, в каталоге файловой системы — информация о находящихся в нем файлах.

Что касается компьютерных сетей (локальных или глобальных), здесь также уместно говорить о каталогах, содержащих объекты разных типов: зарегистрированные пользователи, доступные сетевые принтеры и очереди печати и т. д. Для пользователей сети важно уметь находить и использовать такие

объекты (а их в крупной сети может быть огромное количество), администраторы же сети должны поддерживать эти объекты в работоспособном состоянии. Под *службой каталога* (directory service) понимается та часть распределенной компьютерной системы (компьютерной сети), которая предоставляет средства для поиска и использования имеющихся сетевых ресурсов. Другими словами, служба каталога — это единое образование, объединяющее данные об объектах сети и совокупность служб, осуществляющих манипуляцию этими данными.

В гетерогенной (неоднородной) компьютерной сети могут одновременно функционировать несколько различных служб каталогов, например, локальный диспетчер SAM (Security Account Manager, диспетчер учетных записей безопасности) для компьютеров, не входящих в домен, Windows Directory Service для домена Windows NT 4.0 и Active Directory для Windows 2000/2003/2008. Естественно, для прямого доступа к разным службам каталогов приходится использовать разные инструментальные средства, что усложняет процесс администрирования сети в целом. Для решения этой проблемы можно применить технологию ADSI (Active Directory Service Interface) фирмы Microsoft, которая предоставляет набор объектов ActiveX, обеспечивающих единообразный, не зависящий от конкретного сетевого протокола, доступ к функциям различных каталогов. Объекты ADSI включаются в операционные системы Windows, начиная с Windows 2000.

Для того чтобы находить объекты в каталоге по их именам, необходимо определить для этого каталога *пространство имен* (namespace). Скажем, файлы на жестком диске находятся в пространстве имен файловой системы. Уникальное имя файла определяется расположением этого файла в пространстве имен, например:

```
c:\windows\command\command.com
```

Пространство имен службы каталогов также предназначено для нахождения объекта по его уникальному имени, которое обычно определяется расположением этого объекта в каталоге, где он ищется. Разные службы каталогов используют различные виды имен для объектов, которые они содержат. ADSI определяет соглашение для имен, с помощью которых можно однозначно идентифицировать любой объект в гетерогенной сетевой среде. Такие имена называются *строками связывания* (binding string) или *строками ADsPath* и состоят из двух частей. Первая часть имени определяет, к какой именно службе каталогов (или, другими словами, к какому именно *провайдеру ADSI*) мы обращаемся, например:

- `LDAP://` — для службы каталогов, созданной на основе протокола LDAP (Lightweight Directory Access Protocol), в том числе для Active Directory в Windows 2000/2003/2008;

- `WinNT://` — для службы каталогов в сети Windows NT 4.0 или на локальной рабочей станции Windows 2000/XP/Vista.

Вторая часть строки ADsPath определяет расположение объекта в конкретном каталоге. Приведем несколько примеров строк ADsPath:

```
"LDAP://ldapsrv1/CN=Popov,DC=DEV,DC=MSFT,DC=COM"
"LDAP://CN=Popov,OU=IT,DC=SBRM,DC=COM"
"WinNT://Domain1/Server1,Computer"
"WinNT://Domain1/Popov"
```

По аналогии с командлетом `Get-WmiObject`, обеспечивающим доступ к объектам WMI, можно предположить, что обращение к объектам ADSI осуществляет командлет с именем типа `Get-AdsiObject`. Однако такого командлета нет; в Windows PowerShell вообще отсутствует специальный командлет для доступа к объектам ADSI. Вместо этого следует использовать адаптер типов `[ADSI]`, после которого указывается строка связывания, задающая путь к нужному объекту. Например, для подключения к пользователю Popov из подразделения IT домена sbrm.com можно выполнить следующую команду:

```
PS C:\> $user = [ADSI]"LDAP://CN=Popov,OU=IT,DC=SBRM,DC=COM"
```

Для работы с локальными учетными записями нужно подключиться к соответствующему компьютеру, указав в строке связывания его имя. Для подключения к локальному компьютеру в качестве имени достаточно указать точку:

```
PS C:\> $comp = [ADSI]"WinNT://."
```

К сожалению, в PowerShell 1.0 поддержка ADSI реализована не в полной мере. Например, командлет `Get-Member` не показывает, какие свойства и методы доступны для определенного таким образом объекта `$comp`. Тем не менее, пользоваться этими свойствами и методами можно. Для примера создадим с помощью метода `Create` нового локального пользователя `PSUser`:

```
PS C:\> $user = $comp.Create("user","PSUser")
```

Заполним свойство `Description`, содержащее описание пользователя. Для этого нужно вызвать метод `Put` с двумя параметрами: именем атрибута пользователя (в нашем случае это "Description") и значением, которое должно быть присвоено данному атрибуту:

```
PS C:\> $user.Put("Description","Пользователь создан из PowerShell")
```

Теперь нужно сохранить нового пользователя в базе данных локальных учетных записей с помощью метода `SetInfo`:

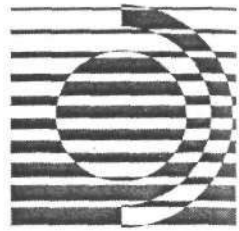
```
PS C:\> $user.SetInfo()
```

Подключимся теперь к новому пользователю и проверим у него значение атрибута `Description`:

```
PS C:\> $user1 = [ADSI]"WinNT://./PSUser,user"
PS C:\> $user1.Description
Пользователь создан из PowerShell
```

Итак, мы убедились, что информация о новом пользователе сохранена в базе данных локальных учетных записей и что к ней можно обратиться из оболочки PowerShell.

Еще раз отметим, что в PowerShell 1.0 работать с ADSI не так просто, как с другими внешними объектами. Это связано с тем, что поддержка интерфейсов ADSI было добавлена в PowerShell 1.0 на довольно поздней стадии разработки, и поэтому не была тщательно проработана. В следующих версиях оболочки этот недостаток должен быть устранен.



Глава 11

Работа с файловой системой

Все люди, работающие с компьютером, ежедневно выполняют множество обращений к файловой системе, читая и сохраняя файлы, создавая каталоги, копируя файлы из одного места в другое и т. п. В данной главе мы научимся манипулировать объектами файловой системы с помощью средств оболочки PowerShell.

Рассмотрим наиболее часто выполняемые операции при работе с файлами и каталогами.

Навигация в файловой системе

В оболочке cmd.exe смена текущего каталога производится с помощью команды `cd`. В PowerShell команда `cd` имеет тот же смысл, при этом она является стандартным псевдонимом командлета `Set-Location`. Например, следующая команда делает текущим каталог c:\windows:

```
PS C:\> cd c:\windows
PS C:\WINDOWS>
```

Как и в оболочке cmd.exe, в качестве пути можно указывать символы .. (для перехода в родительский каталог) и \ (для перехода в корневой каталог текущего диска). При этом нужно учитывать следующий нюанс. Человек, часто пользовавшийся командой `cd` в оболочке cmd.exe, скорее всего, будет машинально набирать команды типа `cd..` или `cd\` без дополнительных пробелов. В PowerShell это вызовет ошибку:

```
PS C:\WINDOWS> cd\
Условие "cd\" не распознано как командлет, функция, выполняемая программа
или файл сценария. Проверьте условие и повторите попытку.
В строка:1 знак:3
+ cd\ <<<<
```

Эта ошибка связана с тем, что в PowerShell параметры команды всегда должны отделяться от имени самой команды пробелом. Поэтому последнюю команду нужно выполнять следующим образом:

```
PS C:\WINDOWS> cd \
```

Получение списка файлов и каталогов

Напомним, что в оболочке cmd.exe список файлов и каталогов формируется с помощью внутренней команды `dir`, которая имеет множество ключей, позволяющих, например, выводить только файлы с определенными атрибутами, обрабатывать вложенные подкаталоги, сортировать список по различным полям и т. д. В PowerShell также можно использовать команду `dir`, которая является здесь псевдонимом командлета `Get-Childitem`. Если выполнить эту команду без параметров, то будет выведено содержимое текущего каталога:

```
PS C:\> dir

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         09.09.2007     21:29            ARCHIV
d----         09.09.2007     21:29            BOOKS
d----         09.09.2007     21:37            Distrib
d----         09.09.2007     21:39            DOC&SMPL
d----         20.11.2005     20:49            Documents and Settings
d----         11.10.2004     18:12            Drivers for notebook
. . .
```

В пути, который указывается для команды `dir`, можно применять подстановочные знаки. Например, следующая команда выведет все файлы с расширением log из каталога c:\windows:

```
PS C:\> dir c:\windows\*.log

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         16.06.2008     11:33          0 0.log
-a---         25.10.2005     16:40        645 cmsetacl.log
```

```
-a---        24.09.2004     20:08       1396 COM+.log
-a---        18.11.2007     10:49     281317 comsetup.log
-a---        24.09.2004     19:31        178 DHCPUPG.LOG
-a---        18.02.2006     18:11        794 DirectX.log
-a---        25.10.2005     17:02        832 DtcInstall.log
. . .
```

Параметр `-Exclude` позволяет задать маску файлов, которые не будут обрабатываться командой `dir`. Например, следующая команда выведет все файлы с расширением log из каталога c:\windows, кроме тех, чье имя начинается на букву d:

```
PS C:\> dir c:\windows\*.log -Exclude d*.log

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---        16.06.2008     11:33          0 0.log
-a---        25.10.2005     16:40        645 cmsetacl.log
-a---        24.09.2004     20:08       1396 COM+.log
-a---        18.11.2007     10:49     281317 comsetup.log
-a---        18.11.2007     10:49     771190 FaxSetup.log
-a---        18.11.2007     10:49    1032996 iis6.log
. . .
```

Параметр `-Name` позволяет выводить на экран только имена файлов (таким образом, этот параметр является аналогом ключа `/b` команды `dir` из cmd.exe), например:

```
PS C:\> dir c:\windows\*.log -Name
0.log
cmsetacl.log
COM+.log
comsetup.log
DHCPUPG.LOG
DirectX.log
DtcInstall.log
FaxSetup.log
iis6.log
imsins.log
. . .
```

Параметр -Recurse включает режим рекурсии, при котором командлет dir отображает не только содержимое указанного каталога, но и всех его подкаталогов:

```
PS C:\> dir 'Documents and Settings' -Recurse

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\Documents and
    Settings

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         18.11.2007     10:51            All Users
d----         20.11.2005     20:52            Tanya
d----         15.06.2008     22:46            Popov

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\Documents and
    Settings\All Users

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d-r--         25.10.2005     16:39            Главное меню
d-r--         11.10.2004     12:01            Документы
d----         21.09.2004     17:53            Избранное
d----         09.03.2008     12:13            Рабочий стол
-a---         20.09.2005     22:45     262144 ntuser.dat
. . .
```

По умолчанию командлет dir не "видит" скрытые файлы. Если необходимо такие файлы также включать в список, то нужно указать параметр -Force:

```
PS C:\> dir -Force

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         09.09.2007     21:29            ARCHIV
d----         09.09.2007     21:29            BOOKS
d----         09.09.2007     21:37            Distrib
d----         09.09.2007     21:39            DOC&SMPL
d----         20.11.2005     20:49            Documents and Settings
d----         11.10.2004     18:12            Drivers for notebook
. . .
```

Иногда бывает нужно сформировать список, состоящий только из подкаталогов или только из файлов. В оболочке cmd.exe подобную фильтрацию в команде `dir` позволяет выполнять ключ `/a`, задающий ограничения на атрибуты элементов, на которые действует команда (например, команда `dir /a:d` отобразит все подкаталоги текущего каталога). В PowerShell построить список подкаталогов можно путем фильтрации объектов с помощью командлета `Where-Object` по значению логического свойства `PSIsContainer` (если объект соответствует каталогу файловой системы, то данное свойство равно `$True`, если объект соответствует файлу, то свойство `PSIsContainer` равно `$False`). Следующая команда выведет список подкаталогов каталога c:\program files:

```
PS C:\> dir 'c:\program files' | Where-Object {$_.PSIsContainer}

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\program files

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         02.10.2004     11:22            ABBYY
d----         02.10.2004     15:31            Aditor
d----         22.09.2004     18:17            Adobe
d----         19.03.2006     23:11            Agnitum
d----         18.02.2006     18:05            Akella Games
d----         19.03.2006     22:50            Common Files
d----         21.09.2004     17:02            ComPlus Applications
d----         05.11.2004     13:35            Creative
d----         14.11.2007      0:14            DrWeb
d----         11.01.2006      0:22            Far
. . .
```

За сортировку возвращаемого списка файлов и каталогов в команде `dir` оболочки cmd.exe отвечает ключ `/o`. Например, команда `dir /o:-d` выдаст список файлов в текущем каталоге, упорядоченный по времени последнего обращения к ним. В PowerShell сортировку списка можно выполнить по любому свойству соответствующих объектов с помощью командлета `Sort-Object`, например:

```
PS C:\> dir | Sort-Object -Descending LastWriteTime

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         12.06.2008     22:32            _Internet
-a---         12.06.2008     20:24       6480 klop.js
```

```
d----          25.05.2008      9:43              WINDOWS
-a---          11.05.2008     22:47        32958 PnP.html
-a---          09.05.2008      9:04         1286 Proc.html
-a---          08.05.2008     20:04         1102 Mem.html
-a---          08.05.2008     19:37          551 mem1.htm
-a---          08.05.2008     19:30          368 styles.css
. . .
```

Определение размера каталогов

Любому каталогу файловой системы в PowerShell соответствует объект типа `System.IO.DirectoryInfo`, который не имеет свойства, в котором хранился бы размер этого каталога. Поэтому для определения размера каталога нужно подсчитать общий размер файлов, записанных в данном каталоге и всех его подкаталогах. Сделать это можно путем суммирования с помощью командлета `Measure-Object` значений свойства `Length` объектов-файлов, поставляемых по конвейеру командлетом `dir` (`Get-ChildItem`). Например, следующая команда показывает размер в байтах каталога c:\program files (свойство `Sum`) и количество файлов и подкаталогов в этом каталоге (свойство `Count`):

```
PS C:\> dir "c:\program files" -Recurse | Measure-Object -Property Length
-Sum


Count    : 15807
Average  :
Sum      : 3663008355
Maximum  :
Minimum  :
Property : Length
```

Если нужно определить размер в мегабайтах или гигабайтах, то значение свойства `Sum` следует разделить на соответствующую константу-суффикс (1Mb или 1Gb):

```
PS C:\> (dir "c:\program files" -Recurse |
>> Measure-Object -Property Length -Sum).Sum/1Mb
>>
3493.31698894501
```

Как видим, результат содержит 11 знаков после запятой. Можно округлить полученное число до двух знаков:

```
PS C:\> $FolderSize=(dir "c:\program files" -Recurse |
>> Measure-Object -Property Length -Sum).Sum/1Mb
```

```
>>
PS C:\> [math]::round($FolderSize,2)
3493.32
```

Создание файлов и каталогов

Создать новый файл или каталог в PowerShell позволяет командлет `New-Item`. Путь к создаваемому элементу указывается в виде значения параметра `-Path`, а в качестве значения параметра `-ItemType` указывается "directory", если нужно создать каталог, и "file", если нужно создать файл. Например, следующая команда создает на диске c: каталог с именем test_folder:

```
PS C:\> New-Item -Path c:\test_folder -Type "directory"

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         16.06.2008     14:36            test_folder
```

Иногда бывает нужно создавать каталоги, в имени которых присутствовала бы текущая дата в определенном формате. Требуемое текстовое представление даты может быть получено с помощью параметра `-uformat` командлета `Get-Date`. Например, следующая команда создаст на диске c: каталог с именем, совпадающим с текущей датой в формате ГГММДД:

```
PS C:\> New-Item -Path c:\ -Name "$(Get-Date -uformat '%y%m%d')" -Type
"directory"

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         17.06.2008      9:53            080617
```

Если в номере года нужно указывать четыре цифры, то следует изменить спецификатор формата на '`%Y%m%d`':

```
PS C:\> New-Item -Path c:\ -Name "$(Get-Date -uformat '%Y%m%d')" -Type
"directory"

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         17.06.2008      9:55            20080617
```

При создании файла в него сразу можно записать строку, указав ее в качестве значения параметра `-Value`, например:

```
PS C:\> New-Item -Path c:\test_file.txt -Type "file" -Value "Test"

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         16.06.2008     15:01          4 test_file.txt
```

Если попытаться назвать создаваемый файл именем уже существующего файла, то возникнет ошибка:

```
PS C:\> New-Item -Path c:\test_file.txt -Type "file" -Value "Test"
New-Item : The file 'C:\test_file.txt' already exists.
В строка:1 знак:9
+ new-item  <<<< -path c:\test_file.txt -type "file" -value "Test"
```

Для перезаписи существующего файла при создании нужно указать параметр `-Force`:

```
PS C:\> New-Item -Path c:\test_file.txt -Type "file" -Value "Test2"
-Force

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         17.06.2008      9:21          5 test_file.txt
```

Чтение и просмотр содержимого файлов

В оболочке cmd.exe имеется команда `type`, которая выводит содержимое текстового файла на экран. В PowerShell команда `type` является псевдонимом командлета `Get-Content` (другие псевдонимы этого же командлета — `cat` и `gc`), предназначенного для построчного считывания содержимого текстового файла с возвращением объекта для каждой строки (при этом строки отображаются на экране). Например:

```
PS C:\> Get-Content c:\windows\win.ini
; for 16-bit app support
[fonts]
[extensions]
[mci extensions]
```

```
[files]
[Mail]
MAPI=1
CMC=1
CMCDLLNAME32=mapi32.dll
CMCDLLNAME=mapi.dll
MAPIX=1
. . .
```

Параметр `-Encoding` командлета `Get-Content` позволяет явно указывать кодировку файла для корректной обработки его содержимого. Допустимые значения данного параметра: `Unicode`, `Byte`, `BigEndianUnicode`, `UTF8`, `UTF7`, `Ascii`.

По умолчанию командлет `Get-Content` считывает все строки из файла; их количество можно ограничить с помощью параметра `-TotalCount`. Например, следующая команда считывает первые пять строк из файла c:\windows\win.ini:

```
PS C:\> Get-Content c:\windows\win.ini -TotalCount 5
; for 16-bit app support
[fonts]
[extensions]
[mci extensions]
```

Результат выполнения командлета `Get-Content` можно сохранять в переменной, обращаясь в дальнейшем к ней, как к массиву строк, например:

```
PS C:\> $f = Get-Content c:\windows\win.ini -TotalCount 5
PS C:\> $f.Length
5
PS C:\> $f[0]
; for 16-bit app support
PS C:\> $f[1]
[fonts]
```

Разумеется, *возвращаемый массив строк можно сразу передавать по конвейеру* для обработки другими командлетами, скажем, для какого-либо преобразования и записи в другой файл (соответствующий пример приведен в следующем разделе).

Запись файлов

Ранее в *главе 5* мы уже рассматривали возможность записи данных во внешние файлы с помощью операторов перенаправления (> и >>) и командлета `Out-File`. При этом командлет `Out-File` пытается форматировать записы-

ваемые в файл объекты. Если нужно просто записать в файл текстовую информацию (без дополнительного форматирования), то лучше воспользоваться командлетом `Set-Content`.

Данные для записи в файл могут задаваться в качестве значения параметра `-Value`. Например, следующая команда записывает в файл c:\test.txt строку "Строка из PowerShell":

```
PS C:\> Set-Content c:\test.txt -Value "Строка из PowerShell"
```

Проверим содержимое файла c:\test.txt:

```
PS C:\> type c:\test.txt
Строка из PowerShell
```

Массив записываемых строк может приниматься по конвейеру от других командлетов. В качестве примера мы считаем пять первых строк из файла c:\windows\win.ini и запишем их в файл c:\num.txt, добавив перед каждой строкой ее порядковый номер. Для этого сначала обнулим переменную `$n` (счетчик строк):

```
PS C:\> $n=0
```

Теперь выполним следующий конвейер команд:

```
PS C:\> Get-Content c:\windows\win.ini -TotalCount 5 | ForEach-Object
{$n++;"$n $_"} | Set-Content c:\num.txt
```

Командлет `Get-Content` в этом конвейере считывает строки из файла c:\windows\win.ini, которые по очереди обрабатываются командлетом `ForEach-Object`. Для каждой обрабатываемой строки значение переменной `$n` увеличивается на единицу, после чего в выходной поток помещается значение, вычисляемое при расширении строки `"$n $_"`. Затем полученная строка передается по конвейеру командлету `Set-Content`, который добавляет ее в файл c:\num.txt.

Убедимся, что в файле c:\num.txt записаны нужные строки:

```
PS C:\> type c:\num.txt
1 ; for 16-bit app support
2 [fonts]
3 [extensions]
4 [mci extensions]
5 [files]
```

Если файл, в который производится запись, уже существует, то командлет `Set-Content` заменит его содержимое. В случае, когда требуется добавить строки в конец существующего файла, следует воспользоваться командлетом `Add-Content`.

Копирование файлов и каталогов

В PowerShell копирование файлов и каталогов осуществляется командлетом `Copy-Item`, имеющим псевдоним `copy`. Путь к копируемым файлам при этом указывается в качестве значения параметра `-Path` (данный параметр используется по умолчанию), а путь к целевому каталогу, в который нужно скопировать файлы, задается значением параметра `-Destination`. Например, следующая команда скопирует файл styles.css с корневого каталога диска c: в каталог c:\test_folder:

```
PS C:\> copy c:\styles.css -Destination c:\test_folder
```

Для того чтобы увидеть результат выполнения команды копирования, нужно указать параметр `-PassThru`:

```
PS C:\> copy c:\styles.css -Destination c:\test_folder -PassThru

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\test_folder

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         08.05.2008     19:30        368 styles.css
```

Если путь к копируемым объектам указывает на каталог, то по умолчанию будет скопирован только этот каталог без своего содержимого (этим PowerShell отличается от большинства других оболочек, в том числе от cmd.exe). Например:

```
PS C:\> copy c:\script -Destination c:\test_folder -PassThru

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\test_folder

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         16.06.2008     16:06            Script
```

Параметр `-Recurse` позволяет копировать содержимое вложенных каталогов, например:

```
PS C:\> copy c:\script -Destination c:\test_folder -Recurse -PassThru

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\test_folder

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         16.06.2008     16:08            Script
```

```
    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\test_folder\Script

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         25.03.2008     17:04       1178 1
-a---         24.03.2008      9:49         56 add.ps1
-a---         24.03.2008     10:19        149 call_ps.bat
-a---         27.03.2008     23:14         69 hello2.ps1
-a---         24.03.2008      9:21        248 test.ps1
d----         16.06.2008     16:08            20080310

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\test_folder\Sc
    ript\20080310

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         10.03.2008     13:39        328 2.ps1
-a---         29.03.2008     10:05        355 2008_03_29.ps1
-a---         10.03.2008     13:39        412 3.ps1
```

Можно копировать не все файлы из каталога, а только соответствующие определенной маске. При этом маску можно указать внутри пути для копирования или в качестве значения параметра `-Include`. Например, следующая команда копирует все файлы с расширением ps1 из каталога c:\script в папку c:\test_folder:

```
PS C:\> copy c:\script\*.ps1  -Destination c:\test_folder -PassThru

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\test_folder

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         24.03.2008      9:49         56 add.ps1
-a---         27.03.2008     23:14         69 hello2.ps1
-a---         24.03.2008      9:21        248 test.ps1
```

Однако если необходимо скопировать файлы по маске из всех вложенных подкаталогов, то одним командлетом `Copy-Item` обойтись не удастся. Предварительно требуемые файлы нужно получить командлетом `Get-ChildItem` (`dir`), а затем передать их командлету `Copy-Item` по конвейеру. Например,

следующая команда копирует все файлы с расширением ps1 из каталога c:\script и всех его подкаталогов в папку c:\test_folder:

```
PS C:\> dir -Recurse -Include *.ps1  c:\script\* | copy -Destination
C:\test_folder -PassThru

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\test_folder

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         10.03.2008     13:39        328 2.ps1
-a---         29.03.2008     10:05        355 2008_03_29.ps1
-a---         10.03.2008     13:39        412 3.ps1
-a---         24.03.2008      9:49         56 add.ps1
-a---         27.03.2008     23:14         69 hello2.ps1
-a---         24.03.2008      9:21        248 test.ps1
```

Команда `copy` оболочки cmd.exe позволяла объединять несколько файлов вместе (конкатенация файлов). В PowerShell объединить файлы можно с помощью командлета `Get-Content` (псевдоним `type`) и перенаправления вывода в результирующий файл. Рассмотрим пример. Создадим файлы 1.txt и 2.txt:

```
PS C:\> New-Item -Path c:\1.txt -Type "file" -Value "File 1"

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         17.06.2008      9:01          6 1.txt
PS C:\> New-Item -Path c:\2.txt -Type "file" -Value "File 2"

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         17.06.2008      9:01          6 2.txt
```

Следующая команда объединяет файлы 1.txt и 2.txt в файл 3.txt:

```
PS C:\> type 1.txt, 2.txt > .\3.txt
```

Проверим содержимое файла 3.txt:

```
PS C:\> type 3.txt
File 1
File 2
```

Как видим, конкатенация файлов 1.txt и 2.txt выполнена успешно.

Переименование и перемещение файлов и каталогов

Переименовать файл или каталог можно с помощью командлета `Rename-Item` (псевдоним `ren`). Значение параметра `-Path` этого командлета задает путь к элементам для переименования, а значение параметра `-NewName` — новое имя. Имена этих параметров можно опускать (в этом случае первым должно указываться значение параметра `-Path`). Например, создадим файл c:\1.tmp и переименуем его в файл 2.tmp:

```
PS C:\> New-Item -Path c:\1.tmp -Type "file"

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         17.06.2008     11:39          0 1.tmp

PS C:\> ren 1.tmp 2.tmp
```

Для того чтобы увидеть результат действия командлета `Rename-Item`, нужно указать параметр `-PassThru`:

```
PS C:\> ren 2.tmp 3.tmp -PassThru

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         17.06.2008     11:39          0 3.tmp
```

В отличие от команды `ren` оболочки cmd.exe командлет `Rename-Item` не поддерживает подстановочные знаки в значении параметров `-Path` или `-NewName`. Поэтому для переименования группы файлов придется выполнить несколько дополнительных действий. В качестве примера мы переименуем все файлы с расширением tmp на диске c:, назначив им новое расширение new.

> ***Замечание***
>
> *Для упрощения кода мы предположим, что имена обрабатываемых файлов не* содержат точек, то есть имен типа file.1.tmp быть не может.

Для этого вначале с помощью командлета `Get-ChildItem` (`dir`) формируется коллекция нужных файлов, каждый элемент которой обрабатывается

командлетом ForEach-Object. Для текущего файла с расширением tmp выделяется его имя без расширения (после вычисления выражения $arr=$_.Name.split(".") имя файла будет храниться в первом элементе массива $arr), составляется новое имя файла ($newname=$arr[0]+".new") и выполняется командлет Rename-Item (ren):

```
PS C:\> dir *.tmp | ForEach-Object {
>> $arr=$_.Name.split(".");
>> $newname=$arr[0]+".new";
>> ren $_.fullname $newname -passthru;
>> }
>>

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         17.06.2008     11:39          0 3.new
. . .
```

Командлет Rename-Item позволяет лишь переименовывать файлы или каталоги, а не перемещать их. Если нужно переместить файл или каталог в другую папку, то следует воспользоваться командлетом Move-Item (псевдоним move). Значение параметра -Path данного командлета задает путь к файлам или каталогам для перемещения (в этом пути допускается использование подстановочных знаков), а значение параметра -Destination — путь к каталогу, куда будут перемещены эти файлы или каталоги. Результат перемещения можно увидеть на экране, указав параметр -PassThru. Например, следующая команда переносит в корневой каталог диска c: каталог c:\test_folder\folder1 со всем его содержимым:

```
PS C:\> Move-Item -Path c:\test_folder\folder1 c:\ -PassThru

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         17.06.2008     12:32            Folder1
```

Удаление файлов и каталогов

Удалять объекты файловой системы можно с помощью командлета Remove-Item (псевдоним del). Значение параметра -Path этого командлета задает путь к удаляемым файлам или каталогам (имя параметра в команде можно не ука-

зывать). В пути допускаются подстановочные символы. Кроме того, командлет `Remove-Item` имеет параметр `-Include`, значение которого задает файлы, на которые будет действовать команда, и параметр `-Exclude`, задающий файлы-исключения, которые удаляться не будут.

Например, следующая команда удалит все файлы с расширением ps1 в каталоге c:\test_folder:

```
PS C:\> del c:\test_folder\*.ps1
```

Если попытаться удалить все файлы в каталоге, имеющем подкаталоги, то система выдаст предупреждение:

```
PS C:\> del c:\test_folder\*

Подтверждение
Элемент в C:\test_folder\20080310 имеет дочерние объекты, и параметр
-Recurse не указан. При продолжении все дочерние объекты будут
удалены вместе с элементом. Вы действительно хотите продолжить?
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех
[T] Приостановить[?] Справка
(значением по умолчанию является "A"):
```

Для удаления без предупреждения всех элементов в каталоге, включая подкаталоги, следует указать параметр `-Recurse`:

```
PS C:\> del c:\test_folder\* -Recurse
```

Поиск текста в файлах

В принципе для поиска текста в файле можно считать его содержимое в массив строк с помощью командлета `Get-Content` и применить к ним один из операторов сравнения, описанных в *главе 7*. Однако в PowerShell имеется командлет `Select-String`, специально предназначенный для поиска строк текста в одном или нескольких файлах (аналогичной утилитой в cmd.exe является findstr).

Например, следующая команда ищет слово Error во всех файлах с расширением log в системном каталоге Windows (имена параметров `-Pattern` и `-Path` можно опустить, не забывая при этом, что шаблон поиска должен стоять после имени командлета первым):

```
PS C:\> Select-String -Pattern Error -Path $env:windir\*.log

WINDOWS\comsetup.log:219:COM+[5:2:56]: Warning: error 0x800704cf in
IsWin2001PrimaryDomainController
```

```
WINDOWS\comsetup.log:493:COM+[7:54:18]: Warning: error 0x80070836
in IsWin2001PrimaryDomainController
WINDOWS\comsetup.log:518:COM+[8:7:42]: ComPlusSetRole: Role = COM+
[8:7:42]: Set users 'COM+[8:7:42]: ERROR: Throwing Exception - FILE:
 d:\nt\com\com1x\src\complussetup\comsetup\csetuputil.cpp, LINE: 8595
WINDOWS\comsetup.log:519:COM+[8:7:42]: RaiseException: Error Code:
0x8011040f
WINDOWS\comsetup.log:521:COM+[8:7:42]: Error 0x8011040f in
HandlePendingInfOperations
. . .
```

Как видим, на экран по умолчанию выводятся: путь к проверяемом файлу, номер строки, в которой найдено совпадение, и сама эта строка. На самом деле результат поиска является объектом типа `Microsoft.PowerShell.Commands.MatchInfo`, содержащим свойства `Path` (путь к файлу), `Pattern` (шаблон, по которому производился поиск), `LineNumber` (номер найденной строки в файле), `Line` (содержимое найденной строки) и другие:

```
PS C:\> Select-String Error $env:windir\*.log | Format-List *

IgnoreCase : True
LineNumber : 219
Line       : COM+[5:2:56]: Warning: error 0x800704cf in
             IsWin2001PrimaryDomainController
Filename   : comsetup.log
Path       : C:\WINDOWS\comsetup.log
Pattern    : Error

IgnoreCase : True
LineNumber : 493
Line       : COM+[7:54:18]: Warning: error 0x80070836 in
             IsWin2001PrimaryDomainController
Filename   : comsetup.log
Path       : C:\WINDOWS\comsetup.log
Pattern    : Error

IgnoreCase : True
LineNumber : 518
Line       : COM+[8:7:42]: ComPlusSetRole: Role = COM+[8:7:42]: Set
             users 'COM+[8:7:42]: ERROR: Throwing Exception - FILE:
             d:\nt\com\com1x\src\complussetup\comsetup\csetuputil.cpp,
```

```
            LINE: 8595
Filename    : comsetup.log
Path        : C:\WINDOWS\comsetup.log
Pattern     : Error
. . .
```

Если при поиске нужно учитывать регистр символов, то следует указать параметр `-CaseSensitive`:

```
PS C:\> Select-String Error $env:windir\*.log -CaseSensitive

WINDOWS\comsetup.log:519:COM+[8:7:42]: RaiseException: Error Code:
0x8011040f
WINDOWS\comsetup.log:521:COM+[8:7:42]: Error 0x8011040f in
HandlePendingInfOperations
WINDOWS\comsetup.log:1188:COM+[11:55:50]: Setup Error: d:\nt\com\
com1x\src\complussetup\comsetup\csetuputil.cpp 9244.
WINDOWS\comsetup.log:1204:COM+[11:56:0]: Setup Error: d:\nt\com\
com1x\src\complussetup\comsetup\csetuputil.cpp 9244.
WINDOWS\KB842773.log:20:13.239: Error getting find handle for
c:\windows.1\$hf_mig$\*.*
. . .
```

По умолчанию командлет `Select-String` находит в файлах все соответствия для строк. В случае необходимости можно ограничиться только первым найденным соответствием, указав параметр `-List`, например:

```
PS C:\> Select-String Error $env:windir\*.log -CaseSensitive -List

WINDOWS\comsetup.log:519:COM+[8:7:42]: RaiseException: Error Code:
0x8011040f
WINDOWS\KB842773.log:20:13.239: Error getting find handle for
c:\windows.1\$hf_mig$\*.*
WINDOWS\setupact.log:1005:BITSINST 2004-10-09 19:19:07  Success
     BITSINST execution has finished. Error code is 0x0.
WINDOWS\spupdsvc.log:341:361.459: [SpUpdDeleteService] DeleteService
Failed: Error:1072
WINDOWS\svcpack.log:34:288.375: Error getting find handle for
c:\windows.1\$hf_mig$\*.*
WINDOWS\tsoc.log:536:hydraoc.cpp(263)Error:StandAlone:TSOC Did not
 get OC_COMPLETE_INSTALLATION.
. . .
```

В результате мы видим, что для каждого файла на экран выводится только по одной строке, содержащей слово Error.

В том случае, когда нужно определить сам факт наличия искомой строки в файлах и неважно, в каком именно файле и на какой позиции находится эта строка, то можно использовать параметр `-Quiet`. В этом командлет возвращает `$True`, если поиск завершен успешно, и `$False` в противном случае:

```
PS C:\> Select-String Error $env:windir\*.log -CaseSensitive -Quiet
True
```

В качестве шаблона для поиска можно использовать регулярные выражения. Например, следующая команда будет искать в файлах с расширением log в системном каталоге Windows строки, содержащие слова Error или Warning:

```
PS C:\> Select-String -Pattern '(Error|Warning)' $env:windir\*.log

WINDOWS.1\comsetup.log:219:COM+[5:2:56]: Warning: error 0x800704cf in
IsWin2001PrimaryDomainController
WINDOWS.1\comsetup.log:493:COM+[7:54:18]: Warning: error 0x80070836 in
IsWin2001PrimaryDomainController
WINDOWS.1\comsetup.log:518:COM+[8:7:42]: ComPlusSetRole: Role = COM+
[8:7:42]: Set users 'COM+[8:7:42]: ERROR: Throwing Exception - FILE:
 d:\nt\com\com1x\src\complussetup\comsetup\csetuputil.cpp, LINE: 8595
WINDOWS.1\comsetup.log:519:     COM+[8:7:42]: RaiseException: Error Code:
0x8011040f
. . .
```

> ***Замечание***
>
> *Подробное рассмотрение языка регулярных выражений выходит за рамки настоящей книги. Дополнительную информацию о некоторых аспектах регулярных выражений можно найти во встроенной справке PowerShell (команда `Get-Help about_regular_expression`); более подробные справочные материалы имеются в библиотеке MSDN.*

Файлы, в которых производится поиск, могут не только указываться в качестве значения параметра `-Path`, но и поступать командлету `Select-String` по конвейеру. Например, следующая команда ищет слово Microsoft во всех файлах с расширением txt, хранящихся в каталоге Windows\System32 и всех его подкаталогах:

```
PS C:\> dir $env:windir\system32\* -Include *.txt -Recurse |
Select-String -Pattern Microsoft -CaseSensitive

WINDOWS.1\system32\drivers\gmreadme.txt:3:Copyright (c) 1998-2000
```

```
Microsoft Corporation.  All Rights Reserved.
WINDOWS.1\system32\drivers\gmreadme.txt:8:Roland GS Sound Set/Microsoft
(P) 1996 Roland Corporation U.S.
WINDOWS.1\system32\drivers\gmreadme.txt:9:The Roland SoundCanvas Sound
Set is licensed under Microsoft's
WINDOWS.1\system32\drivers\gmreadme.txt:10:End User License Agreement for
use with Microsoft operating
WINDOWS.1\system32\windowspowershell\v1.0\ru\about_alias.help.txt:21:
Если в качестве псевдонима для программы Microsoft Word задать слово
. . .
```

Также по конвейеру можно передать массив строк, в котором будет производиться поиск, например:

```
PS C:\> "Windows","PowerShell" | Select-String -Pattern [PR]owerShell

PowerShell
```

Отметим, что в PowerShell 1.0 при поиске символов русского алфавита с помощью командлета `Select-String` могут возникать проблемы, связанные с некорректным определением кодировки текста. В PowerShell 2.0 командлет `Select-String` имеет дополнительный параметр `-Encoding`, позволяющий явно указать нужную кодировку.

Замена текста в файлах

Напомним, что замена текста в строках осуществляется оператором `-replace`, поддерживающим регулярные выражения (см. *главу 7*). Поэтому для замены текста в файле можно считать его содержимое в массив строк с помощью командлета `Get-Content`, обработать нужные строки оператором `-replace` и сохранить результат в том же или другом файле, используя командлет `Set-Content`.

Рассмотрим простой пример. Создадим файл c:\test.txt из трех строк:

```
PS C:\> "11111", "2222", "33333" > c:\test.txt
PS C:\> type c:\test.txt
11111
2222
33333
```

Теперь заменим в этом файле все символы "1" на "a":

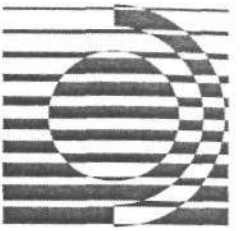
```
PS C:\> (Get-Content C:\test.txt) | ForEach-Object {
>> $_ -replace "1", "a"
```

```
>> } | Set-Content C:\test.txt
>>
```

Первая команда в конвейере здесь взята в скобки, для того чтобы командлет `Set-Content` начал работать лишь после того, как командлет `Get-Content` полностью прочитает файл (в противном случае может возникнуть ошибка при одновременном обращении к файлу c:\test.txt при чтении и записи). Убедимся, что замена прошла успешно:

```
PS C:\> type c:\test.txt
aaaaa
2222
33333
```

Глава 12



Управление процессами и службами

Для обеспечения стабильной и эффективной работы операционной системы большое значение имеет мониторинг работы приложений (запущенных процессов) и настройка функционирования служб (процессов, запускаемых в фоновом режиме). В данной главе мы рассмотрим вопросы, связанные с управлением процессами и службами из оболочки PowerShell.

Управление процессами

Долгое время в операционной системе Windows не было стандартной утилиты, позволяющей из командной строки просматривать список запущенных процессов и управлять ими (подобные инструменты включались в состав пакета Windows Resource Kit для той или иной версии операционной системы). Основным графическим инструментом для управления процессами на локальной машине является Диспетчер задач Windows (для запуска Диспетчера задач можно нажать комбинацию клавиш <Ctrl>+<Shift>+<Esc>), который отображает информацию о выполняющихся процессах, позволяет останавливать их и задавать приоритет выполнения (рис. 12.1).

Начиная с версии Windows XP, в поставки операционных систем включены утилиты командной строки `tasklist` (просмотр списка процессов, запущенных на локальном или удаленном компьютере) и `taskkill` (остановка процесса):

```
PS C:\> tasklist /?

TASKLIST [/S <система> [/U <имя пользователя> [/P [<пароль>]]]]
         [/M [<модуль>] | /SVC | /V] [/FI <фильтр>] [/FO <формат>] [/NH]

Описание:
    Отображает список приложений и связанные с ними задачи/процессы,
```

```
    которые исполняются в текущий момент на локальном или удаленном
    компьютере.
. . .
PS C:\> taskkill /?

TASKKILL [/S <система> [/U <пользователь> [/P [<пароль>]]]]
         { [/FI <фильтр>] [/PID <процесс> | /IM <образ>] } [/F] [/T]

Описание:
    Эта команда позволяет завершить один или несколько процессов.
    Процесс может быть завершен по имени образа или по идентификатору
    процесса.
. . .
```

Разумеется, ими можно пользоваться и в PowerShell, но эта среда предоставляет и более удобные встроенные средства. Рассмотрим способы решения с помощью PowerShell некоторых типичных задач, связанных с процессами.

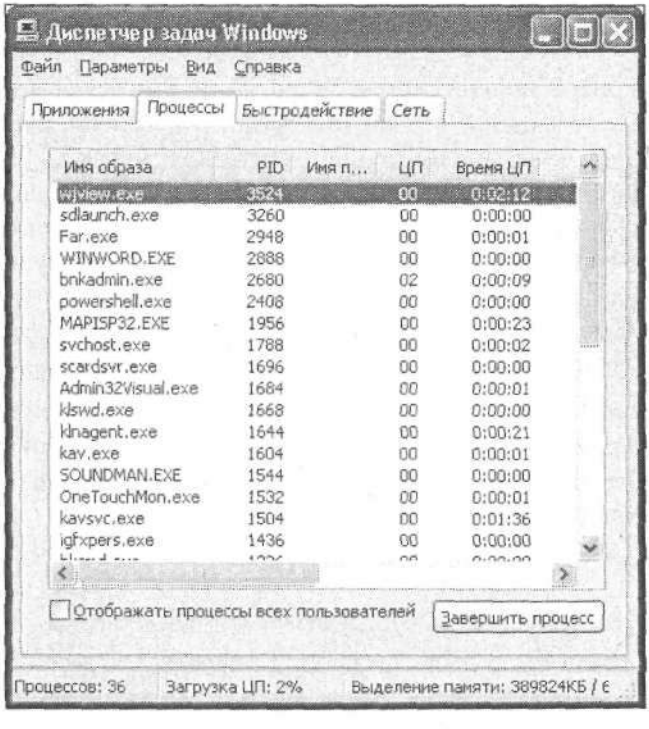


Рис. 12.1. Диспетчер задач Windows

Просмотр списка процессов

В PowerShell получить список запущенных процессов позволяет командлет `Get-Process`. Если запустить этот командлет без параметров, то на экран будет выведена информация обо всех запущенных процессах:

```
PS C:\> Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    109       5     1128        792    32     0.06   1360 alg
    506       6     2308       3604    29    20.34    628 csrss
     67       3      832        436    29     1.52   1736 ctfmon
    553      17    17220      11992    94    69.16   1164 explorer
     33       2     3280        400    35     2.41   3192 Far
     85       3      632        172    21     0.05   1596 hkcmd
      0       0        0         16     0               0 Idle
     90       3      660        176    22     0.02   1608 igfxpers
    133       4     1760        980    37     1.06   1676 kav
. . .
```

Каждому процессу соответствует объект типа `System.Diagnostics.Process`; командлет `Get-Process` по умолчанию отображает несколько свойств этих объектов (табл. 12.1).

Таблица 12.1. *Свойства, отображаемые по умолчанию командлетом* `Get-Process`

Свойство	Описание
`Handles`	Счетчик дескрипторов (свойство `HandleCount` объекта `System.Diagnostics.Process`)
`NPM`	Объем невыгружаемого пула (свойство `NonpagedSystemMemorySize` объекта `System.Diagnostics.Process`)
`PM`	Объем используемой виртуальной памяти (свойство `PagedSystemMemorySize` объекта `System.Diagnostics.Process`)
`WS`	Объем используемой памяти (свойство `WorkingSet` объекта `System.Diagnostics.Process`)
`VM`	Объем выгружаемого пула (свойство `VirtualMemorySize` объекта `System.Diagnostics.Process`)
`CPU`	Количество процессорного времени, затраченное процессом (свойство `TotalProcessorTime.TotalSeconds` объекта `System.Diagnostics.Process`)

Таблица 12.1 *(окончание)*

Свойство	Описание
ID	Идентификатор процесса (свойство Id объекта System.Diagnostics.Process)
ProcessName	Имя процесса (свойство ProcessName объекта System.Diagnostics.Process)

Воспользовавшись параметром -Name (данный параметр используется по умолчанию), можно вывести информацию об одном или нескольких процессах с определенными именами (при этом в именах можно применять подстановочные символы). Например, следующая команда выведет все процессы, имена которых начинаются на букву "s":

```
PS C:\> Get-Process s*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    106       2      380         32    11     0.03   1880 scardsvr
    134       4     1876        200    88     0.11   3640 sdlaunch
    267       6     1504       1236    46     3.67    696 services
     19       1      164         52     4     0.06    560 smss
     71       2     1820        588    30     0.14   1352 SOUNDMAN
    297       6     6924       1768    54     5.05   1304 spoolsv
    134       3     1424        824    33     0.16    876 svchost
    314      13     1768       1280    36     0.75    944 svchost
   1185      45    12124       6532    83    14.59   1040 svchost
     59       3     1124        740    27     1.23   1076 svchost
    224       7     2356        788    38     0.27   1204 svchost
    130       4     2392       1324    35     0.81   1952 svchost
    517       0        0         52     1    26.75      4 System
```

Для сортировки или фильтрации списка процессов применяются, как обычно, командлеты Sort-Object, Where-Object и Select-Object. Например, следующий конвейер команд выводит на экран 5 процессов, которые тратят наибольшее количество процессорного времени:

```
PS C:\> Get-Process | Sort-Object CPU -Descending | Select-Object -First 5

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    619      19    21044      23156   110   171.19   1164 explorer
    790      50    59228      11788   191    93.17   1520 kavsvc
```

```
    476      14    10320    11120    129    88.25   2860 OUTLOOK
    448      19    61204    23412    165    83.59   3884 wjview
    411      12     9780     8920    250    61.95   3484 WINWORD
```

Для просмотра полной информации об определенном процессе можно воспользоваться командлетом `Format-List`, отобразив все свойства объекта `System.Diagnostics.Process`. Например:

```
PS C:\> Get-Process outlook | Format-List *

__NounName                  : Process
Name                        : OUTLOOK
Handles                     : 472
VM                          : 135835648
WS                          : 10010624
PM                          : 10584064
NPM                         : 14552
Path                        : C:\Program Files\Microsoft Office\Office\
                              OUTLOOK.EXE
Company                     : Microsoft Corporation
CPU                         : 88.875
FileVersion                 : 9.0.2416
ProductVersion              : 9.0.2416
Description                 : Microsoft Outlook
Product                     : Microsoft Outlook
Id                          : 2860
PriorityClass               : Normal
HandleCount                 : 472
WorkingSet                  : 10010624
PagedMemorySize             : 10584064
PrivateMemorySize           : 10584064
VirtualMemorySize           : 135835648
TotalProcessorTime          : 00:01:28.8750000
BasePriority                : 8
ExitCode                    :
HasExited                   : False
ExitTime                    :
Handle                      : 2276
MachineName                 : .
MainWindowHandle            : 263442
MainWindowTitle             : Входящие - Microsoft Outlook
```

```
MainModule                    : System.Diagnostics.ProcessModule
                                (OUTLOOK.EXE)
MaxWorkingSet                 : 1413120
MinWorkingSet                 : 204800
Modules                       : { OUTLOOK.EXE, ntdll.dll, kernel32.dll,
                                OUTLLIB.dll...}
NonpagedSystemMemorySize      : 14552
NonpagedSystemMemorySize64    : 14552
PagedMemorySize64             : 10584064
PagedSystemMemorySize         : 109464
PagedSystemMemorySize64       : 109464
PeakPagedMemorySize           : 11055104
PeakPagedMemorySize64         : 11055104
PeakWorkingSet                : 22802432
PeakWorkingSet64              : 22802432
PeakVirtualMemorySize         : 152793088
PeakVirtualMemorySize64       : 152793088
PriorityBoostEnabled          : True
PrivateMemorySize64           : 10584064
PrivilegedProcessorTime       : 00:00:51.0156250
ProcessName                   : OUTLOOK
ProcessorAffinity             : 1
Responding                    : True
SessionId                     : 0
StartInfo                     : System.Diagnostics.ProcessStartInfo
StartTime                     : 24.04.2008 8:05:59
SynchronizingObject           :
Threads                       : { 1756, 316, 3108, 808...}
UserProcessorTime             : 00:00:37.8593750
VirtualMemorySize64           : 135835648
EnableRaisingEvents           : False
StandardInput                 :
StandardOutput                :
StandardError                 :
WorkingSet64                  : 10010624
Site                          :
Container                     :
```

Командлет `Get-Process` позволяет анализировать только процессы, запущенные на локальном компьютере. Для просмотра процессов на удаленных

машинах можно воспользоваться WMI-объектом Win32_Process. Например, следующая команда выведет данные о процессах, запущенных на компьютере \\Popov (по умолчанию информация отображается в виде списка):

```
PS C:\> Get-WmiObject Win32_Process -ComputerName popov
. . .
ProcessName                : System
Handles                    : 376
VM                         : 1925120
WS                         : 28672
Path                       :
__GENUS                    : 2
__CLASS                    : Win32_Process
__SUPERCLASS               : CIM_Process
__DYNASTY                  : CIM_ManagedSystemElement
__RELPATH                  : Win32_Process.Handle="4"
__PROPERTY_COUNT           : 45
__DERIVATION               : { CIM_Process, CIM_LogicalElement,
                             CIM_ManagedSystemElement }
__SERVER                   : POPOV
__NAMESPACE                : root\cimv2
__PATH                     : \\POPOV\root\cimv2:Win32_Process.Handle="4"
Caption                    : System
CommandLine                :
CreationClassName          : Win32_Process
CreationDate               :
CSCreationClassName        : Win32_ComputerSystem
CSName                     : POPOV
Description                : System
ExecutablePath             :
ExecutionState             :
Handle                     : 4
HandleCount                : 376
InstallDate                :
KernelModeTime             : 80515776
MaximumWorkingSetSize      : 1413120
MinimumWorkingSetSize      : 0
Name                       : System
OSCreationClassName        : Win32_OperatingSystem
```

```
OSName                    : Microsoft Windows XP Professional|
                            C:\WINDOWS.1|\Device\Harddisk0\Partition1
OtherOperationCount       : 5499
OtherTransferCount        : 646566
PageFaults                : 4712
. . .
```

Для получения информации о процессах на удаленной машине в виде таблицы нужно воспользоваться командлетом `Format-Table`, указав свойства, которые нужно отобразить. Например, следующая команда выведет таблицу, содержащую данные об именах процессов, их идентификаторах и объеме занимаемой памяти в байтах, упорядочив ее по имени процесса:

```
PS C:\> Get-WmiObject Win32_Process -ComputerName popov | Sort-Object
Name | Format-Table Name, ProcessId, WS

Name                                ProcessId                    WS
----                                ---------                    --
alg.exe                                   252                204800
ati2evxx.exe                             1632                335872
atiptaxx.exe                              400                151552
csrss.exe                                 796               2252800
ctfmon.exe                               1076                434176
explorer.exe                              208               3764224
GoogleToolbarNotifi...                    368               1052672
iexplore.exe                              432              29396992
kav.exe                                  1756               1015808
. . .
```

Определение библиотек, используемых процессом

У объектов `System.Diagnostics.Process`, возвращаемых командлетом `Get-Process`, есть свойство `Modules`, содержащее список динамических библиотек, используемых соответствующими процессами. Для просмотра этого списка можно обрабатывать данные объекты командлетом `Select-Object` с параметром `-ExpandProperty`. Например, следующий конвейер команд выводит список всех динамических библиотек, используемых оболочкой PowerShell (процесс powershell):

```
PS C:\> Get-Process powershell | Select-Object -ExpandProperty Modules |
Format-Table
```

```
Size(K) ModuleName                                          FileName
------- ----------                                          --------
    328 powershell.exe                                      C:...
    708 ntdll.dll                                           C:...
    984 kernel32.dll                                        C:...
    352 msvcrt.dll                                          C:...
     68 ATL.DLL                                             C:...
    576 USER32.dll                                          C:...
    284 GDI32.dll                                           C:...
    688 ADVAPI32.dll                                        C:...
    580 RPCRT4.dll                                          C:...
   1268 ole32.dll                                           C:...
    560 OLEAUT32.dll                                        C:...
    276 mscoree.dll                                         C:...
    472 SHLWAPI.dll                                         C:...
   5508 mscorwks.dll                                        C:...
    620 MSVCR80.dll                                         C:...
   8292 shell32.dll                                         C:...
   1036 comctl32.dll                                        C:...
    616 comctl32.dll                                        C:...
  11160 mscorlib.ni.dll                                     C:...
   7928 System.ni.dll                                       C:...
    560 Microsoft.PowerShell.ConsoleHost.ni.dll             C:...
   5168 System.Management.Automation.ni.dll                 C:...
    160 rsaenh.dll                                          C:...
     96 Microsoft.PowerShell.ConsoleHost.resources.ni.dll   C:...
     96 System.Configuration.Install.dll                    C:...
    536 Microsoft.PowerShell.Commands.Management.ni.dll     C:...
    192 Microsoft.PowerShell.Security.ni.dll                C:...
   1064 Microsoft.PowerShell.Commands.Utility.ni.dll        C:...
   6560 System.Data.ni.dll                                  C:...
   2844 System.Data.dll                                     C:...
     92 WS2_32.dll                                          C:...
     32 WS2HELP.dll                                         C:...
    596 CRYPT32.dll                                         C:...
     72 MSASN1.dll                                          C:...
   5536 System.Xml.ni.dll                                   C:...
   1216 System.DirectoryServices.ni.dll                     C:...
    376 System.Management.dll                               C:...
```

```
264 System.Management.Automation.resources.ni.dll          C:...
640 urlmon.dll                                             C:...
 32 VERSION.dll                                            C:...
 64 Microsoft.PowerShell.Security.resources.ni.dll         C:...
332 mscorjit.dll                                           C:...
 36 shfolder.dll                                           C:...
 88 Microsoft.PowerShell.Commands.Utility.resources.ni.dll C:...
548 diasymreader.dll                                       C:...
 44 psapi.dll                                              C:...
300 MSCTF.dll                                              C:...
```

Остановка процессов

Остановить процесс на локальной машине позволяет командлет `Stop-Process`, имеющий псевдоним `kill`. При этом по умолчанию используется параметр `Id`, требующий указания идентификатора останавливаемого процесса (напомним, что идентификатор процесса можно узнать с помощью командлета `Get-Process`). Например, следующая команда остановит процесс с идентификатором 764:

```
PS C:\> Stop-Process 764
```

Для остановки процесса с определенным именем нужно воспользоваться параметром `-Name` командлета `Stop-Process`. Например, следующая команда остановит *все* процессы с именем notepad:

```
PS C:\> Stop-Process -Name notepad
```

Параметр `-Confirm` командлета `Stop-Process` включает режим подтверждения при остановке процессов. При этом на экране отображается как имя, так и идентификатор останавливаемого процесса, например:

```
PS C:\> Stop-Process -Name notepad -Confirm

Подтверждение
Вы действительно хотите выполнить это действие?
Выполнение операции "Stop-Process" над целевым объектом "notepad (384)".
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех
[T] Приостановить  [?] Справка
(значением по умолчанию является "A"):a

Подтверждение
Вы действительно хотите выполнить это действие?
Выполнение операции "Stop-Process" над целевым объектом "notepad (1728)".
```

```
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех
[T] Приостановить  [?] Справка
(значением по умолчанию является "A"):a
```

По умолчанию командлет `Stop-Process` не передает далее по конвейеру объекты, соответствующие останавливаемым процессам, и поэтому на экране ничего не отображается. Для остановки процессов с выводом информации нужно указать параметр `-PassThru`, например:

```
PS C:\> Stop-Process -Name notepad -PassThru

Handles  NPM(K)    PM(K)      WS(K)     VM(M)   CPU(s)     Id  ProcessName
-------  ------    -----      -----     -----   ------     --  -----------
     26       2      624       2172        24               2224 notepad
```

Замечание

Для остановки процессов на удаленных компьютерах можно воспользоваться утилитой командной строки `taskkill` или методом `Terminate()` WMI-объекта `Win32_Process`.

Запуск процессов, изменение приоритетов выполнения

Для запуска процесса из оболочки PowerShell можно просто указать путь к соответствующему исполняемому файлу, например:

```
PS C:\> C:\WINDOWS\system32\calc.exe
```

В результате выполнения этой команды запустится калькулятор Windows. Также запустить процесс можно с помощью командлета `Invoke-Item`, указав в качестве аргумента путь к нужному файлу. Параметр `-Confirm` позволяет запускать процесс только после подтверждения пользователя:

```
PS C:\> Invoke-Item C:\WINDOWS\system32\calc.exe -Confirm

Подтверждение
Вы действительно хотите выполнить это действие?
Выполнение операции "Вызов элемента" над целевым объектом "Элемент:
C:\WINDOWS\system32\calc.exe".
[A] Да  [X] Да для всех  [H] Нет  [B] Нет для всех
[T] Приостановить  [?] Справка
(значением по умолчанию является "A"):a
```

Однако командлет `Invoke-Item` не возвращает объект, соответствующий запущенному процессу, в силу чего мы не можем программно определить,

скажем, идентификатор, назначенный системой данному процессу. Эту проблему можно решить, воспользовавшись методом `Create()` WMI-класса `Win32_Process` (напомним, что с помощью данного метода можно запускать процессы как на локальном, так и на удаленном компьютере). Данный метод является конструктором класса `Win32_Process`, для его вызова можно использовать акселератор типа `[wmiClass]`. Например, запустим вновь калькулятор Windows:

```
PS C:\> $a = ([wmiClass] "Win32_Process").Create("calc.exe")
```

С помощью командлета `Get-Member` посмотрим, какие свойства имеет объект, сохраненный в переменной `$a`:

```
PS C:\> $a | Get-Member

   TypeName: System.Management.ManagementBaseObject#\__PARAMETERS

Name             MemberType Definition
----             ---------- ----------
ProcessId        Property   System.UInt32 ProcessId {get;set;}
ReturnValue      Property   System.UInt32 ReturnValue {get;set;}
__CLASS          Property   System.String __CLASS {get;set;}
__DERIVATION     Property   System.String[] __DERIVATION {get;set;}
__DYNASTY        Property   System.String __DYNASTY {get;set;}
__GENUS          Property   System.Int32 __GENUS {get;set;}
__NAMESPACE      Property   System.String __NAMESPACE {get;set;}
__PATH           Property   System.String __PATH {get;set;}
__PROPERTY_COUNT Property   System.Int32 __PROPERTY_COUNT {get;set;}
__RELPATH        Property   System.String __RELPATH {get;set;}
__SERVER         Property   System.String __SERVER {get;set;}
__SUPERCLASS     Property   System.String __SUPERCLASS {get;set;}
```

Как видим, нам доступно свойство `ProcessId`, в котором хранится идентификатор запущенного процесса. Теперь можно с помощью командлета `Get-Process` получить соответствующий объект типа `System.Diagnostics.Process`:

```
PS C:\> $b = Get-Process -Id $a.ProcessId
PS C:\> $b | Format-List

Id      : 1484
Handles : 28
CPU     : 0.140625
Name    : calc
```

У объекта System.Diagnostics.Process имеется метод set_PriorityClass, позволяющий изменять приоритет выполнения процесса. В качестве параметра данного метода следует указать одну из следующих строк: "RealTime" (реального времени), "High" (высокий), "AboveNormal" (выше среднего), "Normal" (средний), "BelowNormal" (ниже среднего) или "Idle" (низкий). Например, повысим приоритет процесса, которому соответствует объект, сохраненный в переменной $b (метод get_PriorityClass используется для проверки текущего приоритета):

```
PS C:\> $b.get_PriorityClass()
Normal
PS C:\> $b.set_PriorityClass("High")
PS C:\> $b.get_PriorityClass()
High
```

Завершение неотвечающих процессов

У объектов System.Diagnostics.Process, соответствующих запущенным процессам, имеется свойство Responding, которое принимает значение $False, если процесс не реагирует на запросы. Для завершения всех неотвечающих приложений нужно выделить их из общей массы с помощью командлета Where-Object, после чего остановить с помощью командлета Stop-Process:

```
PS C:\> Get-Process | Where-Object {-not $_.Responding} | Stop-Process
```

Управление службами

В операционной системе Windows *служба* (service) — это процесс, который запускается на машине в фоновом режиме для выполнения определенных действий в ответ на запросы пользователей. В качестве примера приведем службу World Wide Web (WWW), работающую на серверах, на которых установлен пакет Internet Information Services (IIS). Служба WWW работает в фоновом режиме на сервере, ожидая получения HTTP-запросов от web-браузеров. При получении такого запроса служба WWW отвечает на него, посылая запрошенный файл или выполняя определенное действие.

Состав имеющихся в наличии и запущенных служб зависит от версии операционной системы и установленных в ней приложений.

Основным инструментом для администрирования служб в графическом режиме на локальном компьютере является консоль **Службы**, которая находится в программной группе **Администрирование** Панели управления (рис. 12.2).

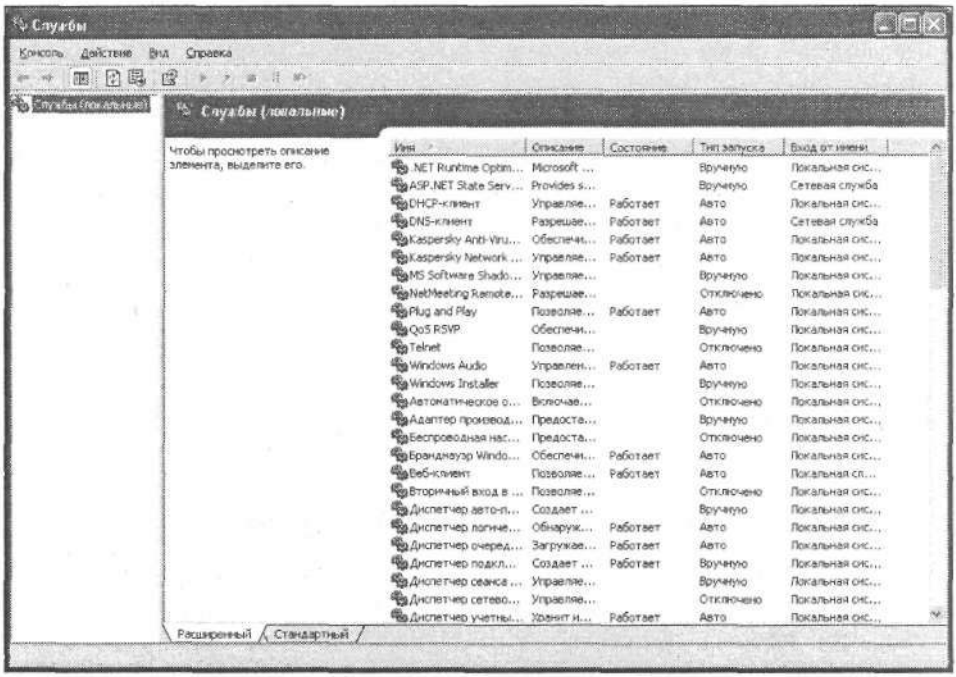


Рис. 12.2. Консоль для управления службами

В качестве утилиты, с помощью которой можно управлять службами как на локальной, так и на удаленных машинах, используется консоль **Управление компьютером**. Также в состав всех версий операционной системы, начиная с Windows NT, входят стандартная утилита net.exe командного интерпретатора cmd.exe, в которой для администрирования служб предусмотрены команды `net start` и `net stop`.

Рассмотрим способы решения с помощью PowerShell некоторых типичных задач, связанных со службами.

Просмотр списка служб

Получить список служб, зарегистрированных на локальном компьютере, позволяет командлет `Get-Service`:

```
PS C:\> Get-Service

Status   Name               DisplayName
------   ----               -----------
Running  Alerter            Оповещатель
Running  ALG                Служба шлюза уровня приложения
```

```
Stopped  AppMgmt          Управление приложениями
Stopped  aspnet_state     ASP.NET State Service
Running  AudioSrv         Windows Audio
Stopped  BITS             Фоновая интеллектуальная служба пер...
Stopped  Browser          Обозреватель компьютеров
Stopped  cisvc            Служба индексирования
Stopped  ClipSrv          Сервер папки обмена
. . .
```

Как видим, по умолчанию отображаются имя службы (колонка `Name`), ее отображаемое имя (колонка `DisplayName`) и состояние (колонка `Status`).

Если нужно вывести на экран только работающие в данный момент службы, то нужно отфильтровать объекты, у которых значением свойства `Status` является строка "Running":

```
PS C:\> Get-Service | Where-Object {$_.Status -eq "Running"}

Status   Name               DisplayName
------   ----               -----------
Running  Alerter            Оповещатель
Running  ALG                Служба шлюза уровня приложения
Running  AudioSrv           Windows Audio
Running  CryptSvc           Службы криптографии
Running  DcomLaunch         Запуск серверных процессов DCOM
Running  Dhcp               DHCP-клиент
Running  dmserver           Диспетчер логических дисков
Running  Dnscache           DNS-клиент
Running  Eventlog           Журнал событий
. . .
```

Для просмотра служб, зарегистрированных на удаленном компьютере, можно воспользоваться классом `Win32_Service` инфраструктуры WMI. Если с помощью командлета `Select-Object` получить свойства `State`, `Name` и `DisplayName` объекта `Win32_Service`, то формат выводимых данных будет похож на формат вывода командлета `Get-Service`. Например:

```
PS C:\> Get-WmiObject -Class Win32_Service -ComputerName Popov |
>> Select-Object -Property State, Name, DisplayName
>>

State                Name                 DisplayName
-----                ----                 -----------
Stopped              Alerter              Оповещатель
Running              ALG                  Служба шлюза уровня...
```

```
Stopped    AppMgmt          Управление приложен...
Stopped    aspnet_state     ASP.NET State Service
Running    Ati HotKey Poller Ati HotKey Poller
Running    AudioSrv         Windows Audio
Running    BITS             Фоновая интеллектуа...
Running    Browser          Обозреватель компью...
Stopped    cisvc            Служба индексирования
. . .
```

Остановка и приостановка служб

Локальную службу можно оставить с помощью командлета `Stop-Service`. Параметр `-Name` задает имя службы для остановки (здесь можно использовать подстановочные знаки), параметр `-Force` позволяет остановить указанную службу вместе со всеми службами, зависящими от нее. Например, следующая команда остановит службу `Spooler` (очередь печати) со всеми зависящими от нее службами:

```
PS C:\> Stop-Service -Name spooler -Force
```

По умолчанию командлет `Stop-Service` не передает далее по конвейеру объекты, соответствующие останавливаемым службам, и поэтому на экране ничего не отображается. Для остановки служб с выводом информации нужно указать параметр `-PassThru`, например:

```
PS C:\> Stop-Service -Name spooler -Force -PassThru

Status   Name               DisplayName
------   ----               -----------
Stopped  Spooler            Диспетчер очереди печати
```

Командлет `Suspend-Service` позволяет приостановить работу одной или нескольких служб, имена которых задаются в качестве значения параметра `-Name`. При приостановке (временной остановке) службы она продолжает выполняться, однако ее действия приостанавливаются до поступления команды на возобновление работы (см. *следующий раздел*). Следует учитывать, что не всякую службу можно приостановить. У объектов `System.ServiceProcess.ServiceController`, соответствующих службам, имеется логическое свойство `CanPauseAndContinue`, которое равно `$True`, если служба может быть приостановлена. Следующая команда приостановит все службы, для которых это возможно:

```
PS C:\> Get-Service | Where-Object {$_.CanPauseAndContinue} |
>> Suspend-Service -PassThru
>>
```

```
Status   Name               DisplayName
------   ----               -----------
Paused   Irmon              Монитор инфракрасной связи
Paused   lanmanserver       Сервер
Paused   lanmanworkstation  Рабочая станция
Paused   Schedule           Планировщик заданий
Paused   seclogon           Вторичный вход в систему
Paused   ShellHWDetection   Определение оборудования оболочки
Paused   TapiSrv            Телефония
Paused   winmgmt            Инструментарий управления Windows
```

Замечание

Для остановки или приостановки служб на удаленном компьютере можно воспользоваться соответственно методами `StopService` и `PauseService` WMI-класса `Win32_Service`.

Запуск и перезапуск служб

Запустить службу на локальном компьютере можно с помощью командлета `Start-Service`. В качестве значения параметра `Name` указывается имя запускаемой службы. Как и в предыдущих командлетах `*-Service`, после запуска службы на экран не выводится никакого сообщения; для вывода информации можно использовать параметр `-PassThru`. Например, следующая команда запускает службу Spooler и выводит на экран информацию об этой службе:

```
PS C:\> Start-Service -Name spooler -PassThru

Status   Name               DisplayName
------   ----               -----------
Running  Spooler            Диспетчер очереди печати
```

Командлет `Restart-Service` выполняет перезапуск (то есть остановку и последующий запуск) служб, указанных в качестве значения параметра `-Name` или полученных по конвейеру. Например, следующий конвейер команд перезапускает все сетевые службы, имя которых начинается со строки "net":

```
PS C:\> Get-Service -Name net* | Restart-Service
```

Изменение параметров службы

В PowerShell имеется командлет `Set-Service`, позволяющий редактировать некоторые параметры служб: отображаемое имя (параметр `-DisplayName`), описание (параметр `-Description`) и тип запуска (параметр `-StartupType`).

Возможные значения последнего параметра — Automatic (служба запускается автоматически), Manual (служба запускается вручную) или Disabled (служба отключена). Имена изменяемых служб указываются в качестве значения параметра -Name (в именах могут присутствовать подстановочные знаки).

Отметим, что ни описание службы, ни тип запуска не являются свойствами объектов System.ServiceProcess.ServiceController, возвращаемых командлетом Get-Service. Другими словами, командлет Get-Service не позволяет проверить, чему равны значения данных параметров для определенной службы. Решить эту проблему можно с помощью WMI-класса Win32_Service, который имеет свойства Description (описание службы) и StartMode (режим запуска службы). Посмотрим, например, чему равно значение этих свойств для службы Schedule (планировщик заданий)

```
PS C:\> Get-WmiObject Win32_Service -Filter "name = 'Schedule'" |
Format-List StartMode, Description

StartMode   : Auto
Description : Позволяет настраивать расписание автоматического выполнения
              задач на этом компьютере. Если эта служба остановлена,
              эти задачи не могут быть запущены в установленное
              расписанием время. Если эта служба отключена, любые службы,
              которые явно зависят от нее, не могут быть запущены.
```

Выведенная информация показывает, что служба планировщика заданий запускается автоматически (значение свойства StartMode равно "Auto"). Изменим с помощью командлета Set-Service режим запуска данной службы с автоматического на ручной:

```
PS C:\> Set-Service -Name Schedule -StartupType Manual
```

Еще раз обратимся к объекту Win32_Service для проверки сделанных изменений:

```
PS C:\> Get-WmiObject Win32_Service -Filter "name = 'Schedule'" | Format-
List Name, StartMode

Name      : Schedule
StartMode : Manual
```

Как видим, значение свойства StartMode успешно изменено.

Глава 13

Работа с системным реестром

Системный реестр Windows — это унифицированная база данных, содержащая информацию об аппаратной и программной конфигурации локального компьютера. Многие настройки системы возможно выполнить, лишь используя реестр, поэтому работать с ним часто бывает необходимо.

Структура реестра

Логически реестр — это древовидная иерархическая база данных. Каждый узел этого дерева называется *разделом* или *ключом* реестра. Реестр напоминает файловую систему: каждый раздел может содержать вложенные разделы (аналог вложенных каталогов) и *параметры* (аналог файлов) (рис. 13.1).

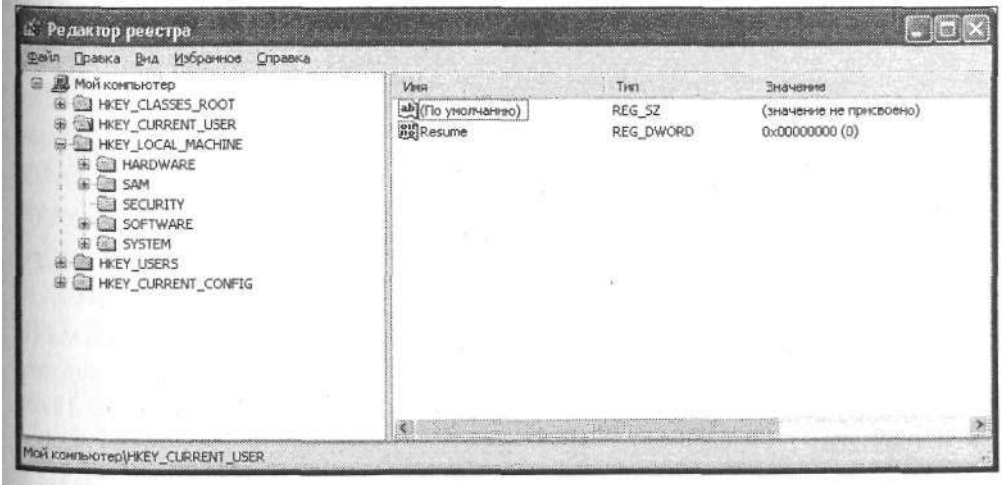


Рис. 13.1. Иерархическая структура системного реестра, отображаемая в Редакторе реестра

Основными в реестре являются 5 корневых разделов:

- HKEY_CLASSES_ROOT (HKCR) содержит сведения о COM-объектах и ассоциациях файлов с приложениями;
- HKEY_LOCAL_MACHINE (HKLM) хранит сведения о рабочей станции, драйверах и системных настройках, то есть информацию, специфичную для компьютера, а не для пользователя;
- HKEY_CURRENT_CONFIG (HKCC) включает в себя информацию о текущей конфигурации аппаратуры компьютера и используется в основном на компьютерах с несколькими аппаратными конфигурациями;
- HKEY_USERS (HKU) содержит данные обо всех пользователях, зарегистрированных на рабочей станции. Здесь хранится информация о каждом пользователе (значения по умолчанию для программ, конфигурации рабочего стола и т. п.);
- HKEY_CURRENT_USER (HKCU) содержит настройки системы и программ, относящиеся к активному пользователю.

Основным графическим средством для просмотра и редактирования системного реестра является программа regedit.exe (на рис. 13.1 показано окно данной программы).

Начиная с версии Windows XP в состав операционной системы включена утилита командной строки reg.exe, которая позволяет добавлять, редактировать, удалять и искать разделы и параметры реестра, выполнять их резервное копирование и восстановление:

```
PS C:\> reg.exe /?
Программа редактирования системного реестра из командной строки, версия 3.0
(C) Корпорация Майкрософт, 1981-2001. Все права защищены

REG <Операция> [Список параметров]

  <Операция> ==  [ QUERY   | ADD     | DELETE  | COPY    |
                   SAVE    | LOAD    | UNLOAD  | RESTORE |
                   COMPARE | EXPORT  | IMPORT ]

Код возврата: (за исключением REG COMPARE)

  0 - Успешно
  1 - С ошибкой
```

Утилиту reg.exe можно применять в командных файлах, автоматизируя наиболее часто выполняемые операции с реестром.

Однако оболочка PowerShell предлагает для работы с системным реестром Windows свои средства, к рассмотрению которых мы и переходим.

Просмотр локального реестра

Доступ к системному реестру на локальном компьютере в PowerShell осуществляется по аналогии с файловой системой путем обращения к соответствующим виртуальным дискам. Если посмотреть с помощью командлета `Get-PSDrive` список доступных дисков, то мы увидим, что корневому разделу (кусту) HKEY_CURRENT_USER соответствует диск HKCU, а корневому разделу (кусту) HKEY_LOCAL_MACHINE — диск HKLM:

```
PS C:\> Get-PSDrive

Name       Provider      Root                 Current Location
----       --------      ----                 ----------------
A          FileSystem    A:\
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
E          FileSystem    E:\
Env        Environment
Function   Function
HKCU       Registry      HKEY_CURRENT_USER
HKLM       Registry      HKEY_LOCAL_MACHINE
Variable   Variable
```

Перейти на диск, соответствующий разделу реестра, можно с помощью командлета `Set-Location` (псевдоним `cd`):

```
PS C:\> cd hklm:
PS HKLM:\>
```

Диски файловой системы содержат папки и файлы. Диски реестра содержат разделы, подразделы и параметры реестра. Выполним команду `Get-ChildItem` (псевдоним `dir`), находясь на диске реестра HKLM:

```
PS HKLM:\> dir

   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE
```

```
SKC   VC   Name                           Property
---   --   ----                           --------
  4    0   HARDWARE                       {}
  1    0   SAM                            {}
Get-ChildItem : Requested registry access is not allowed.
В строка:1 знак:3
+ dir <<<<
 43    1   SOFTWARE                       {(default)}
  8    0   SYSTEM                         {}
```

Как видим, на экран выводятся названия подразделов корневого раздела (колонка `Name`), значения параметров в этих подразделах (колонка `Property`), а также количество вложенных подразделов (колонка `SKC`) и количество параметров внутри разделов (колонка `VC`).

Проверим теперь, показываются ли командлетом `Get-ChildItem` параметры реестра и их значения. Для этого перейдем в раздел HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run, в котором обычно содержатся несколько параметров, определяющих приложения, которые запускаются автоматически при загрузке системы:

```
PS HKLM:\> cd SOFTWARE\Microsoft\Windows\CurrentVersion\Run
PS HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run>
```

Выполним команду `dir` для просмотра дочерних элементов данного раздела:

```
PS HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run> dir

   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\
         Microsoft\Windows\CurrentVersion\Run

SKC  VC Name                           Property
---  -- ----                           --------
  3   0 OptionalComponents             {}
```

Как видим, на экране отображается только вложенный подраздел OptionalComponents и отсутствует информация о параметрах, находящихся внутри раздела. Дело в том, что параметры реестра, в отличие от файлов, нельзя рассматривать в качестве дочерних элементов определенного контейнера, так как в этом случае невозможно обеспечить уникальность имен компонентов реестра (например, путь к разделу реестра с именем Run и путь к параметру реестра (Default) во вложенном разделе Run были бы одинаковыми). Поэтому параметры реестра и их значения не включаются в иерархию разделов и подразделов системного реестра (разделы реестра представлены экземплярами .NET-класса `Microsoft.Win32.RegistryKey`,

а параметры реестра — экземплярами класса System.Management.Automation.PSCustomObject). Вместо этого они считаются свойствами родительских разделов реестра. Для просмотра параметров реестра нужно пользоваться не командой dir (командлет Get-ChildItem), а специальным командлетом Get-ItemProperty (мы уже применяли ранее этот командлет для получения сведений об объектах файловой системы). Выполним командлет Get-ItemProperty для текущего раздела реестра, путь к которому (значение параметра -Path) задается с помощью одной точки:

```
PS HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run> Get-ItemProperty -Path .

PSPath          : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\
                  SOFTWARE\Microsoft\Windows\CurrentVersion\Run
PSParentPath    : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\
                  SOFTWARE\Microsoft\Windows\CurrentVersion
PSChildName     : Run
PSDrive         : HKLM
PSProvider      : Microsoft.PowerShell.Core\Registry
igfxtray        : C:\WINDOWS\system32\igfxtray.exe
igfxhkcmd       : C:\WINDOWS\system32\hkcmd.exe
igfxpers        : C:\WINDOWS\system32\igfxpers.exe
SoundMan        : SOUNDMAN.EXE
KAVWks50        : "C:\Program Files\Kaspersky Lab\Kaspersky Anti-Virus
                  5.0 for Windows Workstations\kav.exe" /minimize /chkas
(default)       :
IndexSearch     : C:\Program Files\Scansoft\PaperPort\IndexSearch.exe
OneTouch Monitor: "C:\Program Files\Xerox One Touch\OneTouchMon.exe"
```

Как видим, в данном случае на экран выводятся стандартные свойства раздела реестра (PSPath, PSParentPath, PSChildName, PSDrive, PSProvider), а также названия и значения всех параметров системного реестра, находящихся внутри раздела SOFTWARE\Microsoft\Windows\CurrentVersion\Run.

Параметр -Name командлета Get-ItemProperty позволяет задавать имена элементов, которые должны быть извлечены из реестра, при этом в именах можно использовать шаблонные символы.

Просмотр удаленного реестра

Получить доступ к системному реестру на удаленной машине не так просто, как на локальном компьютере. Для этого придется напрямую обращаться к объектам .NET или WMI.

Рассмотрим вариант, использующий платформу .NET. Первое, что нужно сделать — определить с помощью статического класса `Microsoft.Win32.RegistryHive` переменную (назовем ее `$type`), определяющую необходимый корневой раздел (куст) реестра. Например, для подключения к разделу HKEY_LOCAL_MACHINE нужно выполнить следующую команду:

```
PS C:\> $type=[Microsoft.Win32.RegistryHive]::LocalMachine
```

Для подключения к другим разделам вместо LocalMachine в данной команде нужно указывать другие имена (ClassesRoot для подключения к HKEY_CLASSES_ROOT, CurrentUser для подключения к HKEY_CURRENT_USER, CurrentConfig для подключения к HKEY_CURRENT_CONFIG и Users для подключения к HKEY_USERS).

Теперь с помощью статического метода `OpenRemoteBaseKey` можно подключиться к корневому разделу удаленного компьютера. В качестве первого параметра данного метода указывается определенная выше переменная `$type`, а в качестве второго параметра — имя компьютера, к которому производится подключение:

```
PS C:\> $RegKey= [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey( $type,
"popov")
```

После выполнения данной команды в переменной `$RegKey` содержится объект класса `Microsoft.Win32.RegistryKey`, соответствующий корневому разделу HKEY_LOCAL_MACHINE реестра на компьютере `\\popov`. Посмотрим с помощью командлета `Get-Member`, какие свойства и методы имеет этот объект:

```
PS C:\> $RegKey | Get-Member

   TypeName: Microsoft.Win32.RegistryKey

Name                      MemberType Definition
----                      ---------- ----------
Close                     Method     System.Void Close()
CreateObjRef              Method     System.Runtime.Remoting.ObjR...
CreateSubKey              Method     Microsoft.Win32.RegistryKey ...
DeleteSubKey              Method     System.Void DeleteSubKey(Str...
DeleteSubKeyTree          Method     System.Void DeleteSubKeyTree...
DeleteValue               Method     System.Void DeleteValue(Stri...
Equals                    Method     System.Boolean Equals(Object...
Flush                     Method     System.Void Flush()
GetAccessControl          Method     System.Security.AccessContro...
GetHashCode               Method     System.Int32 GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeSer...
```

```
GetSubKeyNames              Method      System.String[] GetSubKeyNam...
GetType                     Method      System.Type GetType()
GetValue                    Method      System.Object GetValue(Strin...
GetValueKind                Method      Microsoft.Win32.RegistryValu...
GetValueNames               Method      System.String[] GetValueName...
get_Name                    Method      System.String get_Name()
get_SubKeyCount             Method      System.Int32 get_SubKeyCount...
get_ValueCount              Method      System.Int32 get_ValueCount(...
InitializeLifetimeService   Method      System.Object InitializeLife...
OpenSubKey                  Method      Microsoft.Win32.RegistryKey ...
SetAccessControl            Method      System.Void SetAccessControl...
SetValue                    Method      System.Void SetValue(String ...
ToString                    Method      System.String ToString()
Name                        Property    System.String Name {get;}
SubKeyCount                 Property    System.Int32 SubKeyCount {ge...
ValueCount                  Property    System.Int32 ValueCount {get...
```

Список подразделов можно получить с помощью метода `GetSubKeyNames`:

```
PS C:\> $RegKey.GetSubKeyNames()
HARDWARE
SAM
SECURITY
SOFTWARE
SYSTEM
```

Для получения имен и значений параметров в определенном разделе реестра нужно сначала при помощи метода `OpenSubKey` подключиться к этому разделу. Как и в случае с локальным реестром, мы будем работать с разделом SOFTWARE\Microsoft\Windows\CurrentVersion\Run (соответствующий объект сохраняем в переменной `$RegKey1`):

```
PS C:\> $RegKey1 = $RegKey.OpenSubKey(
>> "SOFTWARE\Microsoft\Windows\CurrentVersion\Run")
>>
```

С помощью метода `GetSubKeyNames` посмотрим, какие подразделы содержит раздел SOFTWARE\Microsoft\Windows\CurrentVersion\Run:

```
PS C:\> $RegKey1.GetSubKeyNames()
OptionalComponents
```

Список параметров, содержащихся в разделе, позволяет получить метод `GetValueNames`:

```
PS C:\> $RegKey1.GetValueNames()
igfxtray
```

```
igfxhkcmd
igfxpers
IndexSearch
```

Значение определенного параметра реестра извлекается с помощью метода GetValue; для формирования списка параметров вместе с их значениями передадим формируемые методом GetValueNames строки по конвейеру командлету Select-Object, на выходе которого будут создаваться объекты с двумя свойствами: Name и Value. Значения свойства Name будем принимать по конвейеру, а значения свойства Value — формировать с помощью метода GetValue:

```
PS C:\> $RegKey1.GetValueNames() | Select-Object @{Name="Name";
Expression={$_}}, @{Name="Value";Expression={$regKey1.getValue($_)}}
Name                 Value
----                 -----
igfxtray             C:\WINDOWS\system32\igfxtray.exe
igfxhkcmd            C:\WINDOWS\system32\hkcmd.exe
igfxpers             C:\WINDOWS\system32\igfxpers.exe
IndexSearch          C:\Program Files\Scansoft\PaperPort\IndexSea...
```

Итак, требуемый список параметров реестра на удаленном компьютере получен.

Модификация реестра

Для реестра на локальном компьютере в PowerShell поддерживаются все командлеты вида *-Item (Get-Item, Copy-Item, Rename-Item и т. д.), позволяющие манипулировать разделами и подразделами реестра, а также командлеты вида *-ItemProperty (Get-ItemProperty, Copy-ItemProperty, Rename-ItemProperty и т. д.) для работы с параметрами реестра. Для изменения реестра на удаленном компьютере можно воспользоваться методами .NET-класса Microsoft.Win32.RegistryKey (процесс подключения к удаленному реестру описан в предыдущем разделе).

Рассмотрим несколько типичных задач, связанных с изменением данных в локальном реестре. Мы будем работать с разделом реестра HKEY_CURRENT_USER\Environment, в котором хранятся имена и значения переменных среды активного пользователя:

```
PS C:\> cd hkcu:
PS HKCU:\Environment> Get-ItemProperty -Path .

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\
               Environment
```

```
PSParentPath: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName : Environment
PSDrive     : HKCU
PSProvider  : Microsoft.PowerShell.Core\Registry
TEMP        : C:\Documents and Settings\404_Popov\Local Settings\Temp
TMP         : C:\Documents and Settings\404_Popov\Local Settings\Temp
```

В нашем случае изначально HKEY_CURRENT_USER\Environment содержит параметры TEMP и TMP, соответствующие одноименным переменным среды.

Создание нового раздела

Создать новый раздел можно с помощью командлета `New-Item`, в качестве значения параметра `Path` которого следует указать путь к данному разделу. Следующая команда создает раздел HKEY_CURRENT_USER\Environment\TestKey:

```
PS HKCU:\Environment> New-Item -Path HKCU:\Environment\TestKey


    Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Environment

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 TestKey                        {}
```

Копирование разделов

Копировать разделы позволяет командлет `Copy-Item`. Следующая команда создает копию раздела HKEY_CURRENT_USER\Environment\TestKey под именем HKEY_CURRENT_USER\Environment\TestCopy:

```
PS HKCU:\Environment> Copy-Item -Path HKCU:\Environment\TestKey
.\TestCopy
```

Проверим содержимое раздела HKEY_CURRENT_USER\Environment:

```
PS HKCU:\Environment> Get-ChildItem


    Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Environment

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 TestCopy                       {}
  0   0 TestKey                        {}
```

Как видим, раздел HKEY_CURRENT_USER\Environment теперь содержит два подраздела.

Переименование раздела

Переименовать раздел можно с помощью командлета `Rename-Item`. В качестве значения параметра `-Path` данного командлета указывается путь к нужному разделу, в качестве значения параметра `-NewName` — новое имя раздела. Например:

```
PS HKCU:\Environment> Rename-Item -Path HKCU:\Environment\TestCopy
-NewName RenamedTest
```

После выполнения этой команды название раздела TestCopy изменится на RenamedTest:

```
PS HKCU:\Environment> Get-ChildItem

   Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Environment

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 RenamedTest                    {}
  0   0 TestKey                        {}
```

Удаление раздела

Удалять разделы позволяет командлет `Remove-Item`, в качестве значения параметра `-Path` которого указывается путь к удаляемым разделам. Например, следующая команда удалит подразделы RenamedTest и TestKey в разделе HKEY_CURRENT_USER\Environment:

```
PS HKCU:\Environment> Remove-Item -Path HKCU:\Environment\Renamedtest,
HKCU:\Environment\TestKey
```

Создание параметра

Для работы с параметрами реестра используются командлеты вида `*-ItemProperty`. Создать новый параметр в разделе можно с помощью командлета `New-ItemProperty`. Имя создаваемого параметра реестра передается в параметре `-Name`, а значение — в параметре `-Value`. Например, следующая команда создаст в разделе HKEY_CURRENT_USER\Environment новый символьный параметр TestParam со значением "TestValue":

```
PS HKCU:\Environment> New-ItemProperty -Path HKCU:\Environment -Name
"TestParam" -Value "TestValue"
```

```
PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
                \Environment
PSParentPath  : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName   : Environment
PSDrive       : HKCU
PSProvider    : Microsoft.PowerShell.Core\Registry
TestParam     : TestValue
```

Проверим содержимое раздела Environment:

```
PS HKCU:\Environment> Get-ItemProperty -Path .

PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
                \Environment
PSParentPath  : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName   : Environment
PSDrive       : HKCU
PSProvider    : Microsoft.PowerShell.Core\Registry
TEMP          : C:\Documents and Settings\404_Popov\Local Settings\Temp
TMP           : C:\Documents and Settings\404_Popov\Local Settings\Temp
TestParam     : TestValue
```

Как видим, новый параметр TestParam создан.

Изменение значения параметра

Изменить значение параметра реестра можно с помощью командлета `Set-ItemProperty`, указав параметры командлета `-Path` (путь к разделу реестра), `-Name` (имя параметра) и `-Value` (новое значение). Например:

```
PS HKCU:\Environment> Set-ItemProperty -Path .\ -Name TestParam -Value
"Updated value"
```

После выполнения данной команды значением параметра TestParam будет строка "Updated Value":

```
PS HKCU:\Environment> Get-ItemProperty -Path . -Name TestParam

PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
                \Environment
PSParentPath  : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName   : Environment
PSDrive       : HKCU
PSProvider    : Microsoft.PowerShell.Core\Registry
TEMP          : C:\Documents and Settings\404_Popov\Local Settings\Temp
TMP           : C:\Documents and Settings\404_Popov\Local Settings\Temp
TestParam     : Updated value
```

Переименование параметра

Переименовать значение параметра реестра можно с помощью командлета `Rename-ItemProperty` с параметрами `-Path` (путь к разделу реестра), `-Name` (старое имя) и `-NewName` (новое имя). Например:

```
PS HKCU:\Environment> Rename-ItemProperty -Path .\ -Name TestParam
-NewName NewParam
```

После выполнения данной команды имя параметра TestParam изменится на NewParam:

```
PS HKCU:\Environment> Get-ItemProperty -Path .

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
               \Environment
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName  : Environment
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
TEMP         : C:\Documents and Settings\404_Popov\Local Settings\Temp
TMP          : C:\Documents and Settings\404_Popov\Local Settings\Temp
NewParam     : Updated value
```

Копирование параметров

Командлет `Copy-ItemProperty` позволяет скопировать параметр реестра из одного раздела в другой (создать копию параметра в том же разделе нельзя). В параметре `-Path` командлета `Copy-ItemProperty` указывается путь к разделу, содержащему копируемый параметр реестра, в параметре `-Destination` — путь к разделу, в который нужно скопировать параметр реестра, в параметре `-Name` — имя копируемого параметра реестра. Например, создадим раздел HKEY_CURRENT_USER\Environment\TestKey и скопируем в него параметр NewParam:

```
PS HKCU:\Environment> New-Item -Path HKCU:\Environment\TestKey

   Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Environment

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 TestKey                        {}
PS HKCU:\Environment> Copy-ItemProperty -Path .\ -Destination .\TestKey
-Name NewParam
```

Убедимся, что параметр NewParam скопирован в раздел HKEY_CURRENT_USER\Environment\TestKey:

```
PS HKCU:\Environment> Get-ItemProperty -Path HKCU:\Environment\TestKey

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
               \Environment\TestKey
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
               \Environment
PSChildName  : TestKey
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
NewParam     : Updated value
```

Очистка значения параметра

Очистить значение определенного параметра реестра можно с помощью командлета `Clear-ItemProperty` с параметрами `-Path` (путь к разделу реестра) и `-Name` (имя параметра реестра). Например:

```
PS HKCU:\Environment> Clear-ItemProperty -Path .\ -Name NewParam
```

Данная команда очищает значение параметра `NewParam` из раздела реестра `HKEY_CURRENT_USER\Environment`:

```
PS HKCU:\Environment> Get-ItemProperty -Path . -Name NewParam

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
               \Environment
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName  : Environment
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
NewParam     :
```

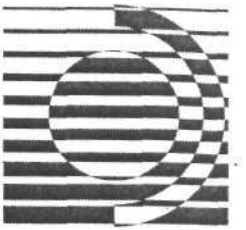
Удаление параметра

Удалить параметр реестра позволяет командлет `Remove-ItemProperty` с параметрами `-Path` (путь к разделу, содержащему удаляемый параметр реестра) и `-Name` (имя удаляемого параметра реестра). Например:

```
PS HKCU:\Environment> Remove-ItemProperty -Path .\ -Name NewParam
```

После выполнения данной команды параметр NewParam будет удален из раздела HKEY_CURRENT_USER\Environment:

```
PS HKCU:\Environment> Get-ItemProperty -Path .

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
               \Environment
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
PSChildName  : Environment
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
TEMP         : C:\Documents and Settings\404_Popov\Local Settings\Temp
TMP          : C:\Documents and Settings\404_Popov\Local Settings\Temp
```



Глава 14

Работа с журналами событий

Под *событием* в Windows понимается любой значительный инцидент в операционной системе или приложении, о котором пользователи (чаще всего администраторы операционной системы) должны быть извещены. Как правило, подобное извещение производится путем записи соответствующей информации в специальные *журналы событий* (Event Logs). События и журналы событий являются важными инструментами администрирования, так как с их помощью можно следить за состоянием операционной системы и исполняемых в ней приложений, анализировать возникающие проблемы. Кроме того, грамотно построенная политика безопасности информационных систем на платформе Windows обязательно должна обеспечивать протоколирование в журналах событий всех наиболее критичных с точки зрения информационной безопасности событий (например, попыток несанкционированного доступа).

Ясно, что администратору системы необходимо иметь простые, гибкие и удобные средства для анализа информации, хранящейся в журналах событий. Также очень важно правильно настроить максимальный размер и режим хранения журналов событий. Журналы могут перезаписываться (новые события будут заменять старые) при достижении определенного размера, однако это может привести к потере важной информации. Лучше отвести для каждого журнала достаточное количество дискового пространства и периодически архивировать и очищать эти журналы, сохраняя тем самым информацию и освобождая дисковое пространство.

На компьютере, работающем под управлением Windows, запись событий производится в следующие журналы.

- **Журнал безопасности** (Security Log) содержит события, которые генерируются в том случае, когда на компьютере настроен аудит. Записи о событиях в журнале безопасности делятся на два типа:
  - *успехи* (Success events) указывают на то, что действие, для которого был настроен аудит, завершилось успешно (например, пользователь

успешно зарегистрировался в сети или успешно получил доступ к файлу на общедоступном ресурсе);

- *отказы* (Failure events) протоколируют, что попытка выполнить действие, для которого был настроен аудит, завершилась безуспешно (например, пользователь попытался зарегистрироваться, однако ввел неправильный пароль, попытался обратиться к сетевому ресурсу, не имея соответствующих разрешений на доступ, и т. п.).

- **Журнал системы** (System Log) хранит события, которые были сгенерированы в результате действий операционной системы (например, запуск служб, сбои в работе драйверов, изменения роли сервера с рядового члена до контроллера домена и т. д.). Записи о событиях системы делятся на три типа:

  - *уведомления* (Information events) просто описывают произведенные действия (например, успешный запуск самой службы Event Log, установление удаленного подключения и т. п.). Некоторые уведомления также содержат информацию о сбоях при выполнении действий, которые реально не влияют на сетевые операции;

  - *предупреждения* (Warning events) содержат информацию о событиях, которые могут стать источником различных проблем (например, сбой динамической регистрации DNS-имен из-за неправильной настройки клиента DNS, сбой службы Windows Time Service при поиске контроллера домена, нехватка пространства на диске и т. п.). Реагировать на подобные предупреждения следует как можно более оперативно;

  - *ошибки* (Error events) сохраняют информацию о критических событиях, которые могут привести к потерям данных или другим серьезным проблемам (сбой при инициализации рабочей станции, отказ в динамическом обновлении со стороны сервера DNS, сбой драйвера устройства и т. п.).

- **Журнал приложений** (Application Log) предназначен для записи событий, сгенерированных запущенными на компьютере приложениями. Для генерации подобных событий разработчики должны вставлять определенный код в свои приложения. Как правило, события приложений оказываются полезными лишь в том случае, когда вы отсылаете информацию об этих событиях разработчикам для решения возникающих проблем. Тем не менее, в журнале приложений также сохраняются некоторые системные события Windows, например, события, возникающие в результате сбоя приложений (они записываются программой Dr. Watson), события, связанные с групповой политикой, нарушения ограничений криптографического экспорта для протокола IPSec, действия IIS, связанные с работой

Active Server Pages (ASP), и т. д. События в журнале приложений также делятся на уведомления, предупреждения и ошибки.

В зависимости от того, какие дополнительные компоненты Windows установлены на компьютере, в системе кроме трех основных журналов могут вестись дополнительные журналы событий:

- **журнал службы каталога** (Directory service log), в который записывается информация о действиях Active Directory. Располагается журнал на контроллерах домена Windows;
- **журнал сервера DNS** (DNS server log), где сохраняется информация о действиях сервера DNS;
- **журнал службы репликации файлов** (File Replication Service log), в котором сохраняются данные о действиях службы File Replication Service (FRS) на тех машинах, где настроена служба DFS (Distributed File System, распределенная файловая система).

Во всех трех перечисленных журналах записи о событиях делятся на уведомления, предупреждения и ошибки.

Кроме этого, собственный журнал событий ведет и сама оболочка Windows PowerShell. В этом журнале регистрируются запуск и остановка интерпретатора PowerShell, а также некоторые ошибки, приводящие к завершению сеанса работы оболочки.

Инструменты для обработки журналов событий

Наиболее привычным и стандартным средством обработки данных в журналах событий является оснастка **Просмотр событий** консоли управления MMC (рис. 14.1).

Она позволяет просматривать произошедшие события (все или удовлетворяющие задаваемому фильтру) на локальном или удаленном компьютере, производить в журнале поиск нужных событий, экспортировать записи журналов в файл с разделителями.

Начиная с Windows XP, те же действия можно выполнять из командной строки с помощью входящей в состав операционной системы VBScript-утилиты eventquery.vbs, в которой применяется технология WMI:

```
PS C:\> cscript c:\windows\system32\eventquery.vbs /?
Сервер сценариев Windows (Microsoft R) версия 5.6
с Корпорация Майкрософт (Microsoft Corp.), 1996-2001. Все права защищены.
```

```
EVENTQUERY.vbs [/S <система> [/U <пользователь> [/P <пароль>]]] [/FI
<фильтр>] [/FO <формат>] [/R <диапазон>] [/NH] [/V] [/L <журнал> | *]

Описание:
    Сценарий EVENTQUERY.vbs позволяет администратору получать нужные
списки событий и свойств из одного или нескольких журналов.
. . .
```

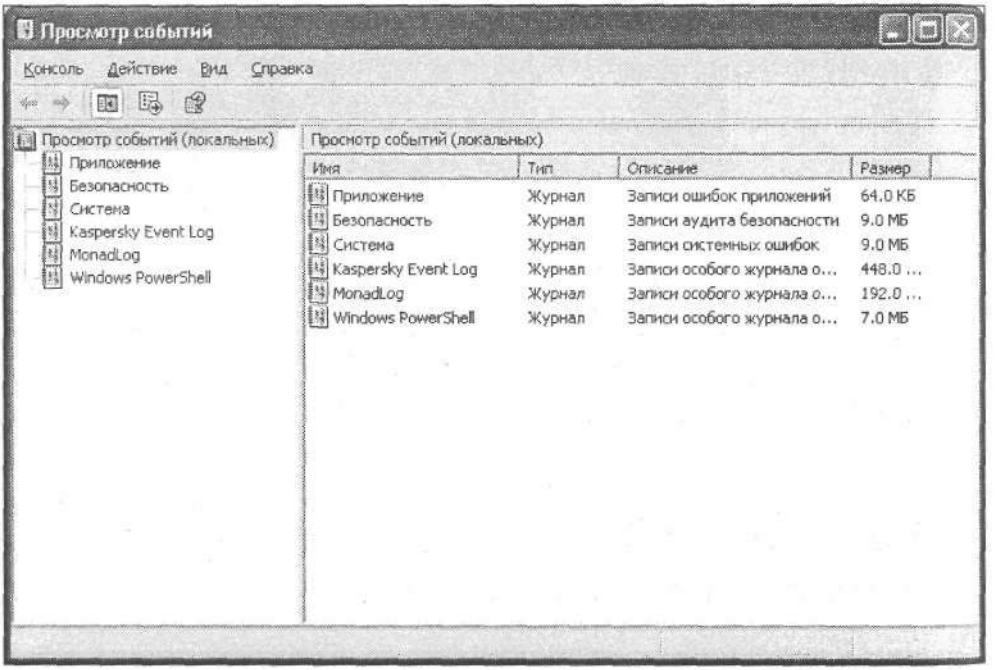


Рис. 14.1. Оснастка **Просмотр событий**

Эта утилита позволяет сохранять выводимую информацию во внешнем файле в форме таблицы или списка с разделителями; применяя ее в командных файлах или сценариях WSH, можно автоматизировать выполнение различных запросов к журналам событий на нескольких машинах.

Однако в PowerShell есть и встроенные средства для работы с журналами событий. Для этих целей служит командлет `Get-EventLog`:

```
PS C:\> Get-Help Get-EventLog

ИМЯ
    Get-EventLog
```

```
ОПИСАНИЕ
    Получает данные о локальных журналах событий или записях, хранимых в
этих журналах.

СИНТАКСИС
    Get-EventLog [-logName] <string> [-newest <int>] [<CommonParameters>]

    Get-EventLog [-list] [-asString] [<CommonParameters>]
. . .
```

С помощью командлета `Get-EventLog` можно вывести список журналов или заданное количество записей из определенного журнала событий.

> ***Замечание***
>
> Командлет `Get-EventLog` может работать только с локальными журналами событий.

Если для решения определенной задачи функциональности данного командлета недостаточно (например, необходимо провести анализ записей из журнала на удаленном компьютере), то можно обращаться непосредственно к соответствующим объектам .NET или WMI (примеры работы с этими объектами приведены далее).

Рассмотрим, каким образом можно из оболочки PowerShell выполнять типичные задачи по обработке журналов событий.

Список журналов событий на локальном компьютере

Вывести список всех доступных журналов событий на локальном компьютере можно с помощью параметра `-List` командлета `Get-EventLog`:

```
PS C:\> Get-EventLog -List

  Max(K) Retain OverflowAction       Entries Name
  ------ ------ --------------       ------- ----
  10 048      0 OverwriteAsNeeded      8 325 Приложение
  15 360      0 OverwriteAsNeeded        258 MonadLog
  10 048      0 OverwriteAsNeeded     32 130 Безопасность
  10 048      0 OverwriteAsNeeded     34 756 Система
  15 360      0 OverwriteAsNeeded      9 712 Windows PowerShell
```

Как видим, по умолчанию система выводит название (отображаемое имя) журнала событий (колонка `Name`), количество записей в этом журнале (колонка `Entries`), информацию о максимальном размере журнала (колонка `Max(K)`) и режиме его заполнения (колонка `OverflowAction`).

В дальнейшем для обращения к журналам событий нам понадобится их краткое название, которое по умолчанию на экране не отображается. Данное название хранится в свойстве `Log` объектов `System.Diagnostics.EventLog`, возвращаемых командлетом `Get-EventLog` (каждому журналу событий соответствует свой объект типа `System.Diagnostics.EventLog`). Для вывода всех свойств этих объектов можно воспользоваться командлетом `Format-List`:

```
PS C:\> Get-EventLog -List | Format-List *

Entries              : {POPOV, POPOV, POPOV, POPOV...}
LogDisplayName       : Приложение
Log                  : Application
MachineName          : .
MaximumKilobytes     : 10048
OverflowAction       : OverwriteAsNeeded
MinimumRetentionDays : 0
EnableRaisingEvents  : False
SynchronizingObject  :
Source               :
Site                 :
Container            :
. . .
Entries              : {POPOV, POPOV, POPOV, POPOV...}
LogDisplayName       : Безопасность
Log                  : Security
MachineName          : .
MaximumKilobytes     : 10048
OverflowAction       : OverwriteAsNeeded
MinimumRetentionDays : 0
EnableRaisingEvents  : False
SynchronizingObject  :
Source               :
Site                 :
Container            :
```

```
. . .
Entries                : {POPOV, POPOV, POPOV, POPOV...}
LogDisplayName         : Система
Log                    : System
MachineName            : .
MaximumKilobytes       : 10048
OverflowAction         : OverwriteAsNeeded
MinimumRetentionDays   : 0
EnableRaisingEvents    : False
SynchronizingObject    :
Source                 :
Site                   :
Container              :
. . .
```

Итак, журналу приложений соответствует название `Application`, журналу системы — `System` и журналу безопасности — `Security`.

Список журналов событий на удаленном компьютере

Как уже отмечалось ранее, командлет `Get-EventLog` может работать только с локальными журналами событий. Для формирования списка журналов событий, ведущихся на удаленном компьютере, можно воспользоваться объектами WMI, которые соответствуют файлам журналов событий (экземпляры класса `Win32_NTEventlogFile`).

> ***Замечание***
>
> Для работы с инфраструктурой WMI на удаленном компьютере необходимо иметь соответствующие права (как правило, права администратора).

Напомним, что подключение к объектам WMI на удаленном компьютере производится командлетом `Get-WmiObject` с параметром `-ComputerName` (имя или IP-адрес компьютера, к которому производится подключение). Например, для получения списка журналов событий на компьютере \\popov нужно выполнить следующую команду:

```
PS C:\> $Logs = Get-WmiObject Win32_NTEventlogFile -ComputerName popov
```

Теперь в переменной `$Logs` содержится нужный нам список журналов. Выведем ее содержимое в виде списка:

```
PS C:\> $Logs | Format-List
```

```
FileSize        : 2359296
LogfileName     : Application
Name            : C:\WINDOWS\system32\config\AppEvent.Evt
NumberOfRecords : 8335

FileSize        : 458752
LogfileName     : Kaspersky Event Log
Name            : C:\WINDOWS\system32\config\kaspersk.evt
NumberOfRecords : 1481

FileSize        : 196608
LogfileName     : MonadLog
Name            : C:\WINDOWS\System32\config\MonadLog.evt
NumberOfRecords : 258

FileSize        : 10027008
LogfileName     : Security
Name            : C:\WINDOWS\System32\config\SecEvent.Evt
NumberOfRecords :

FileSize        : 10289152
LogfileName     : System
Name            : C:\WINDOWS\system32\config\SysEvent.Evt
NumberOfRecords : 34748

FileSize        : 7012352
LogfileName     : Windows PowerShell
Name            : C:\WINDOWS\System32\config\WindowsPower
NumberOfRecords : 9744
```

Как видим, свойство `LogFileName` содержит название журнала, свойство `Name` — путь к файлу журнала событий, свойство `NumberOfRecords` — количество записей в журнале.

Просмотр событий из локального журнала

Для просмотра записей из локального журнала событий проще всего воспользоваться командлетом `Get-EventLog`, указав в качестве значения параметра `-LogName` название нужного журнала событий. Параметр `-Newest` с числовым значением позволяет выделять определенное количество

последних записей из журнала. Например, для вывода последних пяти записей из журнала событий Application нужно выполнить следующую команду:

```
PS C:\> Get-EventLog -Newest 5 -LogName application

Index Time          Type Source                EventID Message
----- ----          ---- ------                ------- -------
 8335 апр 18 22:06  Info MsiInstaller            11707 Product: W...
 8334 апр 18 22:06  Warn WinMgmt                  5603 Поставщик ...
 8333 апр 18 22:06  Warn WinMgmt                  5603 Поставщик ...
 8332 апр 18 20:34  Erro AutoEnrollment             15 Автоматиче...
 8331 апр 18 20:33  Info SecurityCenter           1807 Служба цен...
```

Как обычно, по умолчанию на экран выводится не вся информация, хранящаяся в возвращаемых командлетом объектах. Посмотрим, какие свойства и методы имеются у объектов, соответствующих записям журналов событий. Для этого воспользуемся командлетом `Get-Member`:

```
PS C:\> Get-EventLog -LogName application | Get-Member

   TypeName: System.Diagnostics.EventLogEntry

Name                      MemberType      Definition
----                      ----------      ----------
add_Disposed              Method          System.Void add_Disposed...
CreateObjRef              Method          System.Runtime.Remoting....
Dispose                   Method          System.Void Dispose()
Equals                    Method          System.Boolean Equals(Ev...
GetHashCode               Method          System.Int32 GetHashCode...
GetLifetimeService        Method          System.Object GetLifetim...
GetType                   Method          System.Type GetType()
get_Category              Method          System.String get_Catego...
get_CategoryNumber        Method          System.Int16 get_Categor...
get_Container             Method          System.ComponentModel.IC...
get_Data                  Method          System.Byte[] get_Data()...
get_EntryType             Method          System.Diagnostics.Event...
get_EventID               Method          System.Int32 get_EventID...
get_Index                 Method          System.Int32 get_Index()...
get_InstanceId            Method          System.Int64 get_Instanc...
get_MachineName           Method          System.String get_Machin...
```

```
get_Message                 Method          System.String get_Messag...
get_ReplacementStrings      Method          System.String[] get_Repl...
get_Site                    Method          System.ComponentModel.IS...
get_Source                  Method          System.String get_Source...
get_TimeGenerated           Method          System.DateTime get_Time...
get_TimeWritten             Method          System.DateTime get_Time...
get_UserName                Method          System.String get_UserNa...
InitializeLifetimeService Method            System.Object Initialize...
remove_Disposed             Method          System.Void remove_Dispo...
set_Site                    Method          System.Void set_Site(ISi...
ToString                    Method          System.String ToString()...
Category                    Property        System.String Category {...
CategoryNumber              Property        System.Int16 CategoryNum...
Container                   Property        System.ComponentModel.IC...
Data                        Property        System.Byte[] Data {get;...
EntryType                   Property        System.Diagnostics.Event...
Index                       Property        System.Int32 Index {get;...
InstanceId                  Property        System.Int64 InstanceId ...
MachineName                 Property        System.String MachineNam...
Message                     Property        System.String Message {g...
ReplacementStrings          Property        System.String[] Replacem...
Site                        Property        System.ComponentModel.IS...
Source                      Property        System.String Source {ge...
TimeGenerated               Property        System.DateTime TimeGene...
TimeWritten                 Property        System.DateTime TimeWrit...
UserName                    Property        System.String UserName {...
EventID                     ScriptProperty  System.Object EventID {g...
```

Описание наиболее важных свойств объектов `System.Diagnostics.EventLogEntry`, возвращаемых командлетом `Get-EventLog`, приведено в табл. 14.1.

Таблица 14.1. *Свойства объектов, соответствующих записям журналов событий*

Свойство	Описание
`EventID`	Числовой идентификатор события (например, при успешной регистрации пользователя в системе генерируется запись с идентификатором 528, при отказе в регистрации — запись с идентификатором 529)
`MachineName`	Имя компьютера, на котором произошло событие

Таблица 14.1 (окончание)

Свойство	Описание
Index	Порядковый номер записи в журнале
Category	Категория события (например, "Вход/выход", "Использование прав")
EntryType	Тип события. Для журналов системы и приложений значением данного поля могут быть строки "Information" (информация), "Warning" (предупреждение) или "Error" (ошибка). Для журнала безопасности значением данного поля могут быть строки "FailureAudit" (аудит успехов) и "SuccessAudit" (аудит отказов)
Message	Описание события
Source	Источник события (определенное приложение или компонент системы)
TimeGenerated	Дата и время произошедшего события
UserName	Имя пользователя, к которому относится событие

Журналы событий Windows могут содержать огромное количество записей, большая часть которых отражает нормальную работу системы и не представляет особого интереса. Чтобы не просматривать сотни записей журнала в поисках действительно важных событий, нужно уметь фильтровать события, выбирая записи, удовлетворяющие определенному критерию. Напомним, что в PowerShell для этого применяются специальные командлеты `Where-Object`, `Select-Object` и `ForEach-Object`. Рассмотрим несколько примеров, связанных с фильтрацией записей из журналов событий.

Вывод событий определенного типа

Если нужно отобрать все события определенного типа, то нужно отфильтровать конвейер объектов, возвращаемых командлетом `Get-EventLog` по значению свойства `EntryType`. Например, следующая команда выбирает из пятнадцати последних записей все события, вызвавшие ошибку при работе системы, и выводит результат в виде списка с помощью командлета `Format-List`:

```
PS C:\> Get-EventLog -Newest 15 -LogName system | Where-Object
{$_.EntryType -eq "Error"} | Format-List *

EventID            : 7001
MachineName        : POPOV
Data               : {}
```

```
Index              : 36270
Category           : (0)
CategoryNumber     : 0
EntryType          : Error
Message            : Служба "Обозреватель компьютеров" является зависимой
                     от службы "Сервер", которую не удалось запустить
                     из-за ошибки
                     %Указанная служба не может быть запущена, поскольку
                     она отключена или все связанные с ней устройства
                     отключены.
Source             : Service Control Manager
ReplacementStrings : {Обозреватель компьютеров, Сервер, %%1058}
InstanceId         : 3221232473
TimeGenerated      : 20.04.2008 8:10:57
TimeWritten        : 20.04.2008 8:10:57
UserName           :
Site               :
Container          :
. . .
```

Отбор событий по идентификатору

Администратору операционной системы желательно знать идентификаторы наиболее важных с точки зрения безопасности событий и регулярно анализировать журналы событий на наличие записей с данными идентификаторами. Например, при отказе в регистрации пользователя подсистема безопасности генерирует событие с идентификатором 529 (отказ во входе в систему). Если количество сообщений с данными идентификатором резко увеличилось, это может быть связано с попытками несанкционированного доступа путем подбора паролей. Следующая команда выбирает из ста последних записей в журнале безопасности все события с идентификатором 529 и выводит результат в виде списка с помощью командлета `Format-List` (псевдоним `fl`):

```
PS C:\> Get-EventLog -Newest 100 -LogName security | Where-Object
{$_.EventID -eq 529} | fl *

EventID            : 529
MachineName        : POPOV
Data               : {}
Index              : 32450
Category           : Вход/выход
CategoryNumber     : 2
```

```
EntryType           : FailureAudit
Message             : Отказ входа в систему:
                          Причина:    неизвестное имя пользователя или
                      неверный пароль
                          Пользователь:    popov-av
                          Домен:        SBRM
                          Тип входа:    7
                          Процесс входа:    User32
                          Пакет проверки:    Negotiate
                          Рабочая станция:    POPOV
Source              : Security
ReplacementStrings  : {popov-av, SBRM, 7, User32  ...}
InstanceId          : 529
TimeGenerated       : 20.04.2008 11:23:49
TimeWritten         : 20.04.2008 11:23:49
UserName            : NT AUTHORITY\SYSTEM
Site                :
Container           :
. . .
```

Отбор событий по датам

Дата и время генерации события содержится в свойстве `TimeGenerated` объектов, соответствующих записям журнала событий. Сравнивая значение этого свойства с определенной датой, мы можем выбирать события, произошедшие в определенный день или в каком-то диапазоне дней.

> ***Замечание***
>
> Свойство `TimeGenerated` само является объектом типа `System.DateTime`. В данном типе определены, в частности, свойства для хранения компонентов даты (`Day`, `Month`, `Year`) и времени (`Hour`, `Minute`, `Second`). Сама дата совершения события хранится в поле `Date` объекта `TimeGenerated`.

Например, выведем все сделанные сегодня записи из журнала приложений (напомним, что текущую дату возвращает командлет `Get-Date`):

```
PS C:\> Get-EventLog -LogName application | Where-Object
{$_.TimeGenerated.Date -eq (Get-Date).Date}

Index Time          Type Source                EventID Message
----- ----          ---- ------                ------- -------
 8338 апр 20 08:10  Erro AutoEnrollment             15 Автоматиче...
 8337 апр 20 08:09  Info SecurityCenter           1807 Служба цен...
```

Для просмотра событий, произошедших в течение нескольких последних дней, можно воспользоваться методом AddDays, имеющимся у объектов System.DateTime. Например, следующая команда выведет все записи из журнала событий, сгенерированные не позднее трех дней назад:

```
PS C:\> Get-EventLog -LogName application | Where-Object
{$_.TimeGenerated.Date -ge (Get-Date).AddDays(-3)}

Index Time          Type Source                EventID Message
----- ----          ---- ------                ------- -------
 8338 апр 20 08:10  Erro AutoEnrollment             15 Автоматиче...
 8337 апр 20 08:09  Info SecurityCenter           1807 Служба цен...
 8336 апр 19 03:12  Warn Userenv                  1517 Реестр пол...
 8335 апр 18 22:06  Info MsiInstaller           11707 Product: W...
 8334 апр 18 22:06  Warn WinMgmt                5603 Поставщик ...
 8333 апр 18 22:06  Warn WinMgmt                5603 Поставщик ...
 8332 апр 18 20:34  Erro AutoEnrollment             15 Автоматиче...
 8331 апр 18 20:33  Info SecurityCenter           1807 Служба цен...
 8330 апр 18 20:32  Warn Userenv                  1517 Реестр пол...
```

Чтобы отфильтровать события за определенную дату, можно с помощью параметров -Year, -Month и -Day командлета Get-Date сформировать объект, соответствующий этой дате, и сравнить свойство TimeGenerated с данным объектом. Например, следующая команда выведет все записи из журнала приложений, сгенерированные 18 апреля 2008 года:

```
PS C:\> Get-EventLog -LogName application | Where-Object
{$_.TimeGenerated.Date -eq (Get-Date -Year 2008 -Month 4 -Day 18).Date}

Index Time          Type Source                EventID Message
----- ----          ---- ------                ------- -------
 8335 апр 18 22:06  Info MsiInstaller           11707 Product: W...
 8334 апр 18 22:06  Warn WinMgmt                5603 Поставщик ...
 8333 апр 18 22:06  Warn WinMgmt                5603 Поставщик ...
 8332 апр 18 20:34  Erro AutoEnrollment             15 Автоматиче...
 8331 апр 18 20:33  Info SecurityCenter           1807 Служба цен...
 8330 апр 18 20:32  Warn Userenv                  1517 Реестр пол...
```

Группировка событий по источнику возникновения

С помощью командлета Group-Object удобно анализировать статистику возникновения тех или иных событий. Например, следующая команда показывает, сколько записей в журнале событий системы было сгенери-

ровано различными источниками за последний месяц (группировка объектов-записей журнала событий производится по полю Source):

```
PS C:\> Get-EventLog -LogName system | Where-Object
{$_.TimeGenerated.Date -ge (Get-Date).AddMonths(-1)} | Group-Object source

Count Name                      Group
----- ----                      -----
  358 Service Control Manager   {POPOV, POPOV, POPOV,...
   51 W32Time                   {POPOV, POPOV, POPOV,...
  893 EventLog                  {POPOV, POPOV, POPOV,...
   26 Tcpip                     {POPOV, POPOV, POPOV,...
   10 Application Popup         {POPOV, POPOV, POPOV,...
  305 Print                     {POPOV, POPOV, POPOV,...
    4 NETLOGON                  {POPOV, POPOV, POPOV,...
    2 SRService                 {POPOV, POPOV}
```

Просмотр событий из удаленного журнала

Просматривать содержимое журналов событий на удаленном компьютере можно с помощью объектов .NET или WMI.

Рассмотрим сначала пример, использующий .NET. Подключиться к определенному журналу событий на удаленном компьютере можно путем создания объекта типа System.Diagnostics.EventLog. Для этого нужно вызвать командлет New-Object с параметром -ArgumentList, в качестве значений которого указывается название журнала и имя компьютера, к которому производится подключение. Например, для подключения к журналу событий системы на компьютере \\Popov нужно выполнить следующую команду:

```
PS C:\> $EvLog = New-Object -TypeName System.Diagnostics.EventLog
-ArgumentList System, Popov
```

Свойство Entries объекта System.Diagnostics.EventLog содержит коллекцию объектов System.Diagnostics.EventLogEntry, соответствующих строкам в журнале событий. Чтобы просмотреть несколько последних строк журнала событий, нужно воспользоваться командлетом Select-Object, например:

```
PS C:\> $EvLog.Entries | Select-Object -Last 5 | Format-List *

EventID            : 4201
MachineName        : POPOV
Data               : {0, 0, 0, 0...}
```

```
Index              : 12374
Category           : (0)
CategoryNumber     : 0
EntryType          : Information
Message            : Система обнаружила, что сетевой адаптер Intel(R)
                     ...MiniPCI - Минипорт планировщика пакетов был
                     подключен к сети, и инициировала нормальную работу
                     через этот сетевой адаптер.
Source             : Tcpip
ReplacementStrings : {, Intel(R)...MiniPCI - Минипорт планировщика
                     пакетов}
InstanceId         : 1073746025
TimeGenerated      : 20.04.2008 21:34:58
TimeWritten        : 20.04.2008 21:34:58
UserName           :
Site               :
Container          :
. . .
```

При выводе на экран записей из удаленного журнала событий некоторые сообщения (значения свойства `Message`) могут не отображаться. Это связано с тем, что текст сообщения формируется следующим образом: из записей журнала событий извлекаются строки вставки (значения свойства `ReplacementStrings`), которые вставляются в соответствующие строки формата в библиотеках DLL источника событий, зарегистрированных операционной системой и приложениями. Текст сообщений будет отображаться корректно только тогда, когда на удаленном компьютере разрешен доступ к стандартному административному ресурсу Admin$ и библиотека DLL источника событий находится в каталоге Windows.

Для фильтрации и выделения из общей массы нужных событий можно применять все приемы, описанные в предыдущих разделах (нужно только заменить вызов командлета `Get-EventLog` обращением к свойству `Entries` объекта `System.Diagnostics.EventLog`).

Рассмотрим теперь работу с удаленными журналами событий с помощью объектов WMI. Записям журналов событий соответствуют экземпляры класса `Win32_NTLogEvent`, получить нужные записи можно путем выполнения соответствующего запроса на языке WQL. Например, следующая команда помещает в переменную `$WmiEvLog` указатель на коллекцию всех строк из журнала событий системы на компьютере \\Popov:

```
PS C:\> $WmiEvLog = Get-WmiObject -Query "select * from Win32_NTLogEvent
where (Logfile ='System')" -ComputerName Popov
```

Теперь можно, например, просмотреть несколько последних записей с помощью командлета `Select-Object`:

```
PS C:\> $WmiEvLog | Select-Object -Last 5 | Format-List *

__GENUS          : 2
__CLASS          : Win32_NTLogEvent
__SUPERCLASS     :
__DYNASTY        : Win32_NTLogEvent
__RELPATH        : Win32_NTLogEvent.Logfile="System",RecordNumber=95
                   86
__PROPERTY_COUNT : 16
__DERIVATION     : {}
__SERVER         : POPOV
__NAMESPACE      : root\cimv2
__PATH           : \\POPOV\root\cimv2:Win32_NTLogEvent.Logfile="System",
                   RecordNumber=9586
Category         : 0
CategoryString   :
ComputerName     : POPOV
Data             :
EventCode        : 7036
EventIdentifier  : 1073748860
EventType        : 3
InsertionStrings : {Диспетчер подключений удаленного доступа, Работает}
Logfile          : System
Message          : Служба "Диспетчер подключений удаленного доступа"
                     перешла в состояние Работает.

RecordNumber     : 9586
SourceName       : Service Control Manager
TimeGenerated    : 20080119190722.000000+180
TimeWritten      : 20080119190722.000000+180
Type             : информация
User             :
```

Для фильтрации записей журналов событий, полученных с помощью WMI, можно применять средства WMI (командлеты `Where-Object` и `Select-Object`) или налагать дополнительные условия в WQL-запросе. При больших размерах журналов последнее более эффективно с точки зрения производительности.

Настройка журналов событий

Кроме просмотра и анализа содержимого журналов событий администратору приходится настраивать параметры этих журналов: устанавливать максимальный размер и определять режим хранения.

Установка максимального размера журналов

Максимальный размер журнала можно устанавливать в диапазоне от 64 Кбайт до 4 Гбайт (по умолчанию этот размер равен 512 Кбайт). Для того чтобы не потерять информацию о событиях, следует периодически проверять журналы и увеличивать их максимальный размер в случае, если они растут слишком быстро.

Для примера установим для локального журнала событий приложений максимальный размер 10112 Кбайт. Для этого нужно сначала создать объект `System.Diagnostics.EventLog`, соответствующий журналу `Application`:

```
PS C:\> $EvLog = New-Object -TypeName System.Diagnostics.EventLog
-ArgumentList Application
```

Теперь выполним метод `set_MaximumKilobytes`, указывая требуемый размер журнала в качестве параметра:

```
PS C:\> $EvLog.set_MaximumKilobytes(10112)
```

Максимальный размер журнала событий приложений установлен.

Установка режима хранения журналов

Для настройки периода хранения журнала событий можно выбрать один из трех режимов, каждому из которых в .NET соответствует указанная в скобках символьная константа:

- **затирать старые события по необходимости** (OverwriteAsNeeded) — устанавливается по умолчанию и включает циклическое протоколирование событий. После заполнения журнала старые события будут удаляться, чтобы освободить место для новых (при этом может потеряться важная информация);
- **затирать события, старее определенного числа дней** (OverwriteOlder) — устанавливает другой вид циклического протоколирования событий, при котором перезаписываются лишь достаточно старые события. Данный режим можно выбрать в том случае, когда вы уверены, что максимальный размер журнала достаточно велик для того, чтобы этот журнал не

переполнялся, и вы периодически архивируете и очищаете ваш журнал событий в конце каждого интервала хранения журнала;

- **не затирать события** (DoNotOverwrite) — автоматическая перезапись отсутствует. Этот режим можно использовать в том случае, если у вас достаточно дискового пространства для хранения журнала событий, а безопасность и функционирование системы являются приоритетными задачами. В этом случае необходимо периодически проверять и архивировать журнал событий, после чего очищать события, не допуская его заполнения. В противном случае, после заполнения журнала операционная система Windows перестанет записывать в него новые события.

Для определения текущего режима хранения определенного журнала или установки нового режима нужно предварительно создать объект `System.Diagnostics.EventLog`, соответствующий этому журналу, например:

```
PS C:\> $EvLog = New-Object -TypeName System.Diagnostics.EventLog
-ArgumentList Application
```

Проверить текущий режим хранения позволяет метод `get_OverflowAction`, возвращающий соответствующую символьную константу:

```
PS C:\> $EvLog.get_OverflowAction()
OverwriteAsNeeded
```

Установить новый режим хранения журнала событий можно с помощью метода `ModifyOverflowPolicy`, имеющего два параметра. Первый символьный параметр определяет режим хранения, а второй, числовой, — количество дней хранения записей в режиме OverwriteOlder (для других двух режимов данный параметр должен быть равен нулю). Установим для журнала событий приложений режим, при котором старые события не затираются:

```
PS C:\> $EvLog.ModifyOverflowPolicy("DoNotOverwrite",0)
```

Вызовем теперь метод `get_OverflowAction`:

```
PS C:\> $EvLog.get_OverflowAction()
DoNotOverwrite
```

Как видим, режим хранения журнала приложений действительно изменился.

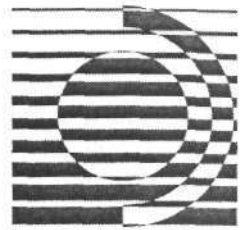
Очистка журнала

Очистить журнал (удалить все записи) можно с помощью метода `Clear` соответствующего объекта `System.Diagnostics.EventLog`. Для примера очистим журнал событий приложений на локальном компьютере:

```
PS C:\> $EvLog = New-Object -TypeName System.Diagnostics.EventLog
-ArgumentList Application
```

```
PS C:\> $EvLog.Clear()
PS C:\> $EvLog

  Max(K) Retain OverflowAction       Entries Name
  ------ ------ --------------       ------- ----
  10 112      0 OverwriteAsNeeded            0 Приложение
```

Как видим, после выполнения метода `Clear` журнал событий не содержит записей (значение свойства `Entries` равно нулю).



Глава 15

Управление рабочими станциями. Получение и анализ системной информации

Подсистема WMI позволяет легко собирать разнообразные сведения о рабочих станциях в сети (системные настройки Windows, состав установленных программных продуктов и обновлений операционной системы, список автоматически запускаемых приложений и т. п.). В этой главе мы рассмотрим примеры получения подобной информации из оболочки PowerShell.

Начнем же мы с того, что научимся с помощью командной строки управлять рабочими станциями: завершать сеансы работы пользователей, перегружать и выключать компьютеры.

Завершение сеанса пользователя

Завершить сеанс работы пользователя можно несколькими способами. Проще всего воспользоваться стандартной утилитой командной строки logoff.exe:

```
PS C:\> logoff /?
Завершение сеанса.

LOGOFF [<имя сеанса> | <ID сеанса>] [/SERVER:<сервер>] [/V]

  <имя сеанса>       Имя сеанса.
  <ID сеанса>        Идентификатор сеанса.
  /SERVER:<сервер>   Сервер терминалов, содержащий завершаемый сеанс
                     пользователя (по умолчанию - текущий).
  /V                 Отображение информации о выполненных действиях.
```

Для завершения сеанса активного пользователя на локальном компьютере достаточно выполнить команду `logoff` без параметров.

Также в Windows имеется утилита shutdown.exe, позволяющая управлять локальной или удаленной рабочей станцией, в том числе завершать сеанс пользователя компьютера:

```
PS C:\> shutdown /?
Использование: C:\WINDOWS\system32\shutdown.exe [-i | -l | -s | -r |
-a] [-f] [-m \\<компьютер>]
                [-t xx] [-c "комментарий"] [-d up:xx:yy]

      Без аргументов      Вывод справки по использованию (как и -?)
      -i                  Отображение графического интерфейса,
                          должен быть первым параметром
      -l                  Выход (не совместим с параметром -m)
      -s                  Завершение работы компьютера
      -r                  Перезагрузка компьютера
      -a                  Прекращение завершения работы системы
      -m \\<компьютер>    Удаленный компьютер, на котором выполняется
                          действие
       -t xx              Тайм-аут завершения работы - xx сек.
       -c "comment"       Комментарий (не более 127 знаков)
       -f                 Принудительное завершение приложений без
                          предварительного предупреждения
       -d [u][p]:xx:yy    Код причины завершения работы
                           u - пользовательский код
                           p - код запланированного завершения
                           xx - основной код причины (1 - 255)
                           yy - дополнительный код причины (1 - 65535)
```

Итак, для завершения сеанса активного пользователя на локальном компьютере можно выполнить команду `shutdown -l`.

Третий вариант завершения сеанса пользователя — вызов метода `Win32Shutdown` WMI-класса `Win32_OperatingSystem` с параметром 0:

```
PS C:\> (Get-WMIObject Win32_OperatingSystem).Win32Shutdown(0)

__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
```

```
__DERIVATION      : {}
__SERVER          :
__NAMESPACE       :
__PATH            :
ReturnValue       : 0
```

Перезагрузка и выключение компьютера

Перезагрузить компьютер из оболочки PowerShell можно несколькими способами.

Во-первых, можно воспользоваться утилитой командной строки tsshutdn.exe:

```
PS C:\> tsshutdn /?
Завершение работы сервера в установленном порядке.

TSSHUTDN [wait_time] [/SERVER:servername] [/REBOOT] [/POWERDOWN]
         [/DELAY:logoffdelay] [/V]

 wait_time              Задержка в секундах после уведомления пользователей
                        до прекращения их сеансов (по умолчанию 60 секунд).
 /SERVER:servername     Завершающий работу сервер (по умолчанию текущий).
 /REBOOT                Перезагрузка сервера после прекращения всех сеансов.
 /POWERDOWN             Подготовка сервера к отключению питания.
 /DELAY:logoffdelay     Задержка в секундах после прекращения всех
                        подключенных сеансов (по умолчанию 30 секунд).
 /V                     Вывод сообщений о выполняемых действиях.
```

Для перезагрузки локального компьютера можно выполнить команду `tsshutdn /reboot`.

Другой способ выполнить перезагрузку компьютера — команда `shutdown -r`.

Перезагрузку рабочей станции можно осуществить и с помощью метода `Reboot` WMI-класса `Win32_OperatingSystem`, выполнив в оболочке PowerShell следующую команду:

```
PS C:\> (Get-WMIObject Win32_OperatingSystem).Reboot()
```

Однако для успешного выполнения метода `Reboot` требуется, чтобы при подключении к подсистеме WMI был активизирован режим использования всех привилегий, что реализовано только в PowerShell версии 2.

Выключить компьютер позволяют те же утилиты командной строки, которые использовались для его перезагрузки (tsshutdn.exe и shutdown.exe), только с другими ключами: `tsshutdn` и `shutdown -s`. Подсистема WMI позво-

ляет выключить компьютер с помощью метода Shutdown класса Win32_OperatingSystem, при этом, как и в случае перезагрузки, должен быть активизирован режим использования всех привилегий.

Получение информации о BIOS

Сведения о системной BIOS компьютера можно получить с помощью WMI-класса Win32_BIOS. Следующая команда выводит на экран значения всех свойств экземпляра данного класса, которые не являются служебными для WMI (названия подобных свойств начинаются с двух символов подчеркивания):

```
PS C:\> Get-WmiObject Win32_BIOS | Select-Object -Property *
-ExcludeProperty __*

Status                : OK
Name                  : EPP runtime BIOS - Version 1.1
Caption               : EPP runtime BIOS - Version 1.1
SMBIOSPresent         : True
BiosCharacteristics   : {4, 7, 8, 9...}
BIOSVersion           : {COMPAQ - 1, EPP runtime BIOS - Version 1.1
                        }
BuildNumber           :
CodeSet               :
CurrentLanguage       : en|US|iso8859-1
Description           : EPP runtime BIOS - Version 1.1
IdentificationCode    :
InstallableLanguages  : 1
InstallDate           :
LanguageEdition       :
ListOfLanguages       : {en|US|iso8859-1}
Manufacturer          : Compaq
OtherTargetOS         :
PrimaryBIOS           : True
ReleaseDate           : 20011129000000.000000+000
SerialNumber          : 7J13FMPZ7062
SMBIOSBIOSVersion     : 1.35
SMBIOSMajorVersion    : 2
SMBIOSMinorVersion    : 3
SoftwareElementID     : EPP runtime BIOS - Version 1.1
SoftwareElementState  : 3
TargetOperatingSystem : 0
Version               : COMPAQ - 1
```

Вывод списка команд, выполняемых при загрузке системы

Все команды, которые выполняются автоматически при старте системы, хранятся в WMI в виде экземпляров класса `Win32_StartupCommand`. В свойстве `Caption` содержатся описания запускаемых программ, в свойстве `Command` хранятся пути к программам и их параметры, а в свойстве `Location` указываются пути к папкам автозагрузки или к разделам реестра, содержащим соответствующие записи:

```
PS C:\> Get-WmiObject Win32_StartupCommand | Format-Table Caption,
Command, Location

Caption                Command                Location
-------                -------                --------
desktop                desktop.ini            Startup
CTFMON.EXE             C:\WINDOWS.1\System...  HKU\S-1-5-18\SOFTWA...
CTFMON.EXE             C:\WINDOWS.1\System...  HKU\S-1-5-19\SOFTWA...
CTFMON.EXE             C:\WINDOWS.1\System...  HKU\S-1-5-20\SOFTWA...
desktop                desktop.ini            Startup
swg                    C:\Program Files\Go...  HKU\S-1-5-21-120266...
ctfmon.exe             C:\WINDOWS.1\system...  HKU\S-1-5-21-120266...
desktop                desktop.ini            Startup
CTFMON.EXE             C:\WINDOWS.1\System...  HKU\.DEFAULT\SOFTWA...
desktop                desktop.ini            Common Startup
InterVideo WinCinem... C:\PROGRA~1\INTERV~...  Common Startup
Microsoft Office       C:\PROGRA~1\MICROS~...  Common Startup
WinampAgent            "C:\Program Files\W...  HKLM\SOFTWARE\Micro...
AtiPTA                 Atiptaxx.exe           HKLM\SOFTWARE\Micro...
Outpost Firewall       C:\Program Files\Ag...  HKLM\SOFTWARE\Micro...
KAVWks50               "C:\Program Files\K...  HKLM\SOFTWARE\Micro...
                                              HKLM\SOFTWARE\Micro...
```

Подробную информацию о командах автозагрузки можно получить с помощью следующей команды (выбираются все свойства объектов `Win32_StartupCommand`, кроме служебных свойств WMI, названия которых начинаются с двух знаков подчеркивания):

```
PS C:\> Get-WmiObject Win32_StartupCommand | Select-Object -Property *
-ExcludeProperty __*

Caption     : desktop
Command     : desktop.ini
```

```
Description : desktop
Location    : Startup
Name        : desktop
SettingID   :
User        : NT AUTHORITY\SYSTEM

Caption     : CTFMON.EXE
Command     : C:\WINDOWS.1\System32\CTFMON.EXE
Description : CTFMON.EXE
Location    : HKU\S-1-5-18\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Name        : CTFMON.EXE
SettingID   :
User        : NT AUTHORITY\SYSTEM

Caption     : CTFMON.EXE
Command     : C:\WINDOWS.1\System32\CTFMON.EXE
Description : CTFMON.EXE
Location    : HKU\S-1-5-19\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Name        : CTFMON.EXE
SettingID   :
User        : NT AUTHORITY\LOCAL SERVICE

Caption     : CTFMON.EXE
Command     : C:\WINDOWS.1\System32\CTFMON.EXE
Description : CTFMON.EXE
Location    : HKU\S-1-5-20\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
Name        : CTFMON.EXE
SettingID   :
User        : NT AUTHORITY\NETWORK SERVICE

Caption     : desktop
Command     : desktop.ini
Description : desktop
Location    : Startup
Name        : desktop
SettingID   :
User        : POPOV\User

Caption     : swg
```

```
Command     : C:\Program Files\Google\GoogleToolbarNotifier\
              GoogleToolbarNotifier.exe
Description : swg
Location    : HKU\S-1-5-21-1202660629-746137067-1708537768-1004\SOFTWARE\
              Microsoft\Windows\CurrentVersion\Run
Name        : swg
SettingID   :
User        : POPOV\User
. . .
```

Вывод свойств операционной системы

Основные свойства установленной операционной системы (загрузочное устройство, номер сборки, дата установки и т. д.) можно получить с помощью экземпляра WMI-класса `Win32_OperatingSystem`. Следующий конвейер команд выводит все свойства данного объекта, кроме служебных свойств WMI:

```
PS C:\> Get-WmiObject Win32_OperatingSystem | Select-Object -Property *
-ExcludeProperty __*

Status                                  : OK
Name                                    : Microsoft Windows XP
                                          Professional|C:\WINDOWS.1|
                                          \Device\Harddisk0\Partition1
FreePhysicalMemory                      : 18240
FreeSpaceInPagingFiles                  : 253980
FreeVirtualMemory                       : 2055224
BootDevice                              : \Device\HarddiskVolume1
BuildNumber                             : 2600
BuildType                               : Uniprocessor Free
Caption                                 : Microsoft Windows XP
                                          Professional
CodeSet                                 : 1251
CountryCode                             : 7
CreationClassName                       : Win32_OperatingSystem
CSCreationClassName                     : Win32_ComputerSystem
CSDVersion                              : Service Pack 2
CSName                                  : POPOV
CurrentTimeZone                         : 240
```

```
DataExecutionPrevention_32BitApplications : False
DataExecutionPrevention_Available         : False
DataExecutionPrevention_Drivers           : False
DataExecutionPrevention_SupportPolicy     : 2
Debug                                     : False
Description                               : Popov
Distributed                               : False
EncryptionLevel                           : 168
ForegroundApplicationBoost                : 2
InstallDate                               : 20040924200342.000000+24
                                            0
LargeSystemCache                          : 0
LastBootUpTime                            : 20080611154729.500000+24
                                            0
LocalDateTime                             : 20080611215711.946000+24
                                            0
Locale                                    : 0419
Manufacturer                              : Microsoft Corporation
MaxNumberOfProcesses                      : 4294967295
MaxProcessMemorySize                      : 2097024
NumberOfLicensedUsers                     :
NumberOfProcesses                         : 29
NumberOfUsers                             : 2
Organization                              :
OSLanguage                                : 1049
OSProductSuite                            :
OSType                                    : 18
OtherTypeDescription                      :
PlusProductID                             :
PlusVersionNumber                         :
Primary                                   : True
ProductType                               : 1
QuantumLength                             : 0
QuantumType                               : 0
RegisteredUser                            : User
SerialNumber                              : 55683-640-7090775-23275
ServicePackMajorVersion                   : 2
ServicePackMinorVersion                   : 0
SizeStoredInPagingFiles                   : 478124
SuiteMask                                 : 272
```

```
SystemDevice                    : \Device\HarddiskVolume1
SystemDirectory                 : C:\WINDOWS\system32
SystemDrive                     : C:
TotalSwapSpaceSize              :
TotalVirtualMemorySize          : 2097024
TotalVisibleMemorySize          : 196080
Version                         : 5.1.2600
WindowsDirectory                : C:\WINDOWS
```

Вывод списка установленных программных продуктов

Каждой программе, которая была установлена на компьютере с помощью Windows Installer, соответствует экземпляр WMI-класса `Win32_Product`. Список всех программ с указанием названия, версии и разработчика можно получить при помощи следующей команды:

```
PS C:\> Get-WmiObject Win32_Product | Format-Table Name, Version, Vendor

Name                    Version          Vendor
----                    -------          ------
MSDN Library - Octo...  7.34.2233        Microsoft
Google Toolbar for ...  4.0.0.002         Google Inc.
MSXML 4.0 SP2 (KB92...  4.20.9841.0       Microsoft Corporation
WMI Tools               1.50.1131.0001   Microsoft Corporation
Adobe Reader 6.0.2 CE   006.000.002      Adobe Systems Incorporated
Microsoft Visio Pro...  10.0.525         Microsoft Corporation
Профессиональный вы...  9.00.2720        Microsoft Corporation
Антивирус Касперско...  5.0.712          Лаборатория Касперского
WebFldrs XP             9.50.5318        Microsoft Corporation
MSN Messenger 6.2       6.2.0137         Microsoft Corporation
Microsoft .NET Fram...  2.0.50727        Microsoft Corporation
```

Если вас интересуют более подробные сведения о программах (например, дата установки или идентификационный номер), то можно вывести все свойства объектов `Win32_Product`, кроме служебных свойств WMI, название которых начинается с двух символов подчеркивания:

```
PS C:\> Get-WmiObject Win32_Product | Select-Object -Property *
-ExcludeProperty __*

Name                : MSDN Library - October 2002
Version             : 7.34.2233
```

```
InstallState      : 5
Caption           : MSDN Library - October 2002
Description       : MSDN Library - October 2002
IdentifyingNumber : {A0484561-DE06-44B3-8D8D-137AEDF28FB3}
InstallDate       : 20040927
InstallDate2      : 20040927000000.000000-000
InstallLocation   :
PackageCache      : C:\WINDOWS.1\Installer\187a1.msi
SKUNumber         :
Vendor            : Microsoft

Name              : Google Toolbar for Internet Explorer
Version           : 4.0.0.002
InstallState      : 5
Caption           : Google Toolbar for Internet Explorer
Description       : Google Toolbar for Internet Explorer
IdentifyingNumber : {DBEA1034-5882-4A88-8033-81C4EF0CFA29}
InstallDate       : 20070414
InstallDate2      : 20070414000000.000000-000
InstallLocation   :
PackageCache      : C:\WINDOWS.1\Installer\2dbf81.msi
SKUNumber         :
Vendor            : Google Inc.

Name              : MSXML 4.0 SP2 (KB927978)
Version           : 4.20.9841.0
InstallState      : 5
Caption           : MSXML 4.0 SP2 (KB927978)
Description       : MSXML 4.0 SP2 (KB927978)
IdentifyingNumber : {37477865-A3F1-4772-AD43-AAFC6BCFF99F}
InstallDate       : 20070207
InstallDate2      : 20070207000000.000000-000
InstallLocation   :
PackageCache      : c:\WINDOWS.1\Installer\c940b.msi
SKUNumber         :
Vendor            : Microsoft Corporation
. . .
```

Класс `Win32_SoftwareFeature` позволяет получить информацию обо всех компонентах программ, установленных с помощью Windows Installer.

Название компонента хранится в свойстве `Caption`, название программы — в свойстве `ProductName`:

```
PS C:\> Get-WmiObject Win32_SoftwareFeature | Format-Table Caption,
ProductName, Vendor, Version

Caption          ProductName      Vendor          Version
-------          -----------      ------          -------
MSDN Library     MSDN Library ... Microsoft       7.34.2233
MSDN Documenta... MSDN Library ... Microsoft      7.34.2233
Platform SDK D... MSDN Library ... Microsoft      7.34.2233
Visual Studio ... MSDN Library ... Microsoft      7.34.2233
Office Develop... MSDN Library ... Microsoft      7.34.2233
Embedded Devel... MSDN Library ... Microsoft      7.34.2233
Developer Know... MSDN Library ... Microsoft      7.34.2233
Enterprise Dev... MSDN Library ... Microsoft      7.34.2233
Windows Develo... MSDN Library ... Microsoft      7.34.2233
XML and Web Se... MSDN Library ... Microsoft      7.34.2233
MSDN Documenta... MSDN Library ... Microsoft      7.34.2233
. . .
```

Все имеющиеся свойства объектов `Win32_SoftwareFeature`, кроме служебных свойств WMI, можно вывести следующим образом:

```
PS C:\> Get-WmiObject Win32_SoftwareFeature | Select-Object -Property *
-ExcludeProperty __*

Status          :
Name            : MSDN_Quarterly_Library
InstallState    : 3
LastUse         : 20040927******.000000+***
Accesses        : 1
Attributes      : 18
Caption         : MSDN Library
Description     : MSDN Library
IdentifyingNumber : {A0484561-DE06-44B3-8D8D-137AEDF28FB3}
InstallDate     :
ProductName     : MSDN Library - October 2002
Vendor          : Microsoft
Version         : 7.34.2233

Status          :
Name            : MSDN_Documentation
```

```
InstallState      : 3
LastUse           : 19800000******.000000+***
Accesses          : 0
Attributes        : 18
Caption           : MSDN Documentation Group 1
Description       : Includes MSDN Architectural Samples Team sample
                    suites, MSDN Magazine, technical articles for all
                    Microsoft technologies, MSDN Online Voices columns,
                    Periodicals
IdentifyingNumber : {A0484561-DE06-44B3-8D8D-137AEDF28FB3}
InstallDate       :
ProductName       : MSDN Library - October 2002
Vendor            : Microsoft
Version           : 7.34.2233

Status            :
Name              : Platform_SDK_Docs_for_VS7
InstallState      : 3
LastUse           : 19800000******.000000+***
Accesses          : 0
Attributes        : 18
Caption           : Platform SDK Documentation
Description       : Platform SDK Documentation
IdentifyingNumber : {A0484561-DE06-44B3-8D8D-137AEDF28FB3}
InstallDate       :
ProductName       : MSDN Library - October 2002
Vendor            : Microsoft
Version           : 7.34.2233
. . .
```

Вывод списка установленных обновлений операционной системы

Для поддержания операционной системы в актуальном состоянии большое значение имеет своевременная установка всех появляющихся обновлений системы безопасности (критических оперативных исправлений (hot fixes) и комплектов исправлений (roll-ups) системы обновлений).

Замечание

Установка обновлений на рабочие станции может производиться вручную или быть организована в автоматическом режиме с помощью служб Automatic Updates, Windows Software Update Services (WSUS) или более мощного средства Microsoft System Management Server (SMS).

Каждому установленному обновлению соответствует экземпляр WMI-класса `Win32_QuickFixEngineering`. Вывести список всех установленных обновлений можно с помощью следующей команды:

```
PS C:\> Get-WmiObject Win32_QuickFixEngineering | Format-Table HotFixID,
FixComments, ServicePackInEffect

HotFixID              FixComments           ServicePackInEffect
--------              -----------           -------------------
File 1                                      KB873339
File 1                                      KB885250
File 1                                      KB885835
File 1                                      KB885836
File 1                                      KB885884
File 1                                      KB886185
File 1                                      KB887472
File 1                                      KB887742
. . .
File 1                                      KB926255
File 1                                      KB929969
Q147222
KB834707-IE6-200409... Update               SP0
Q927978
KB925398_WMP64
KB917734_WMP9
KB923689
KB811113              Service Pack          SP2
KB873339              Update                SP3
KB885250              Update                SP3
KB885835              Update                SP3
KB885836              Update                SP3
KB885884              Update                SP3
. . .
```

Как видим, порядковый номер обновления может содержаться либо в свойстве `HotFixID`, либо в свойстве `ServicePackInEffect` (это обстоятельство следует учитывать, если у вас возникнет необходимость написать сценарий, проверяющий, все ли нужные пакеты обновлений установлены на рабочих станциях).

Свойства объектов `Win32_QuickFixEngineering` позволяют также узнать дополнительную информацию об установленных обновлениях, в частности дату установки и имя пользователя, который ее выполнял. Например, следующая команда выводит подробные данные о трех последних обновлениях:

```
PS C:\> Get-WmiObject Win32_QuickFixEngineering | Select-Object
-Property  * -ExcludeProperty __* -Last 3

Status              :
Caption             :
CSName              : POPOV
Description         : Обновление безопасности для Windows XP (KB925454)
FixComments         : Update
HotFixID            : KB925454
InstallDate         :
InstalledBy         : SYSTEM
InstalledOn         : 1/24/2007
Name                :
ServicePackInEffect : SP3

Status              :
Caption             :
CSName              : POPOV
Description         : Windows PowerShell(TM) 1.0
FixComments         :
HotFixID            : KB926140
InstallDate         :
InstalledBy         :
InstalledOn         :
Name                :
ServicePackInEffect : SP3

Status              :
Caption             :
CSName              : POPOV
Description         : Обновление безопасности для Windows XP (KB929969)
FixComments         : Update
HotFixID            : KB929969
InstallDate         :
InstalledBy         : SYSTEM
InstalledOn         : 1/24/2007
Name                :
ServicePackInEffect : SP3
```

Глава 16

Инвентаризация оборудования

Одна из типичных администраторских задач — инвентаризация оборудования на рабочих станциях в локальной сети. Для автоматизации этого процесса очень хорошо подходят сценарии, которые обращаются к соответствующим объектам WMI. Ранее мы уже неоднократно работали с объектной моделью WMI из оболочки PowerShell — при знании названий нужных классов это совсем несложная задача.

В данной главе мы рассмотрим несколько примеров получения при помощи PowerShell сведений об аппаратных компонентах компьютера и формирования соответствующих отчетов в формате HTML

Получение информации о физической памяти

Получить информацию о банках физической памяти компьютера позволяют экземпляры класса `Win32_PhysicalMemory`. Выведем сначала все свойства этого объекта:

```
PS C:\> Get-WmiObject Win32_PhysicalMemory | Format-List *

__GENUS                : 2
__CLASS                : Win32_PhysicalMemory
__SUPERCLASS           : CIM_PhysicalMemory
__DYNASTY              : CIM_ManagedSystemElement
__RELPATH              : Win32_PhysicalMemory.Tag="Physical Memory 0"
__PROPERTY_COUNT       : 30
__DERIVATION           : {CIM_PhysicalMemory, CIM_Chip,
                         CIM_PhysicalComponent, CIM_PhysicalElement...}
__SERVER               : POPOV
```

```
__NAMESPACE           : root\cimv2
__PATH                : \\POPOV\root\cimv2:Win32_PhysicalMemory.Tag=
                        "Physical Memory 0"
BankLabel             : Bank 0,1: J18
Capacity              : 67108864
Caption               : Физическая память
CreationClassName     : Win32_PhysicalMemory
DataWidth             : 64
Description           : Физическая память
DeviceLocator         : DIMM #1: J18
FormFactor            : 1
HotSwappable          :
InstallDate           :
InterleaveDataDepth   : 0
InterleavePosition    : 0
Manufacturer          :
MemoryType            : 2
Model                 :
Name                  : Физическая память
OtherIdentifyingInfo  :
PartNumber            :
PositionInRow         : 1
PoweredOn             :
Removable             :
Replaceable           :
SerialNumber          :
SKU                   :
Speed                 :
Status                :
Tag                   : Physical Memory 0
TotalWidth            : 64
TypeDetail            : 128
Version               :
. . .
```

Оставим теперь только самые важные свойства и выведем их в табличном виде:

```
PS C:\> Get-WmiObject Win32_PhysicalMemory | Format-Table BankLabel,
Capacity, Description
```

```
BankLabel                    Capacity    Description
---------                    --------    -----------
Bank 0,1: J18                67108864    Физическая память
Bank 2,3: J19               134217728    Физическая память
FLASH ROM: U35                 524288    Физическая память
```

Как видим, объем памяти (колонка `Capacity`) выводится в байтах, что неудобно для визуального просмотра. Преобразуем с помощью командлета `Select-Object` выводящиеся данные таким образом, чтобы значение свойства `Capacity` отображалось в мегабайтах:

```
PS C:\> Get-WmiObject Win32_PhysicalMemory | Select-Object BankLabel,
@{Name="Capacity, Mb"; Expression={$_.Capacity/1Mb}}, Description

BankLabel                    Capacity, Mb    Description
---------                    ------------    -----------
Bank 0,1: J18                          64    Физическая память
Bank 2,3: J19                         128    Физическая память
FLASH ROM: U35                        0.5    Физическая память
```

Преобразование отчета в формат HTML

Преобразуем теперь полученный отчет в формат HTML, чтобы его было удобно просматривать в веб-браузере. Проще всего это сделать с помощью командлета `ConvertTo-HTML` с выводом полученного HTML-кода в файл (командлет `Out-File`):

```
PS C:\> Get-WmiObject Win32_PhysicalMemory | Select-Object BankLabel,
@{Name="Capacity, Mb"; Expression={$_.Capacity/1Mb}}, Description | Con-
vertTo-HTML | Out-File c:\mem.html
```

В результате выполнения этого конвейера команд сформированный отчет будет сохранен в файле c:\mem.html. Для просмотра содержимого данного файла в веб-браузере, используемом по умолчанию, можно воспользоваться командлетом `Invoke-Item`:

```
PS C:\> Invoke-Item c:\mem.html
```

В результате откроется новое окно веб-браузера с HTML-страницей, содержащей нашу таблицу (рис. 16.1).

Заметим, что полученная таблица выглядит не лучшим образом. Командлет `ConvertTo-HTML` генерирует "чистый" HTML-код, без тегов форматирования, поэтому мы можем изменить внешний вид HTML-страницы с помощью внешней таблицы стилей (CSS). В подобных таблицах стилей размещают

правила форматирования различных элементов HTML-документов. Для создания собственной таблицы стилей выполним следующую команду:

```
PS C:\> @'
>> body { background-color:aqua; }
>> body,table,td,th { font-family:Tahoma; color:Black; Font-Size:10pt }
>> th { font-weight:bold; background-color:silver; }
>> td { background-color:white; }
>> '@ > c:\styles.css
>>
```

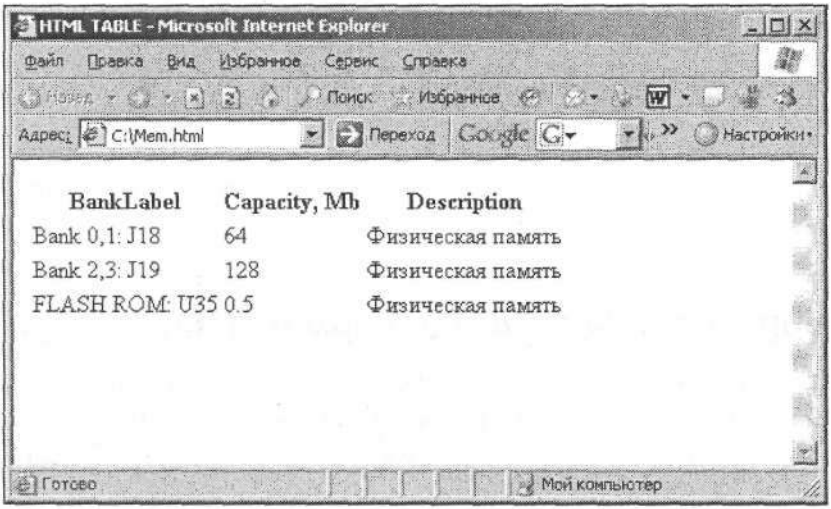


Рис. 16.1. HTML-страница с информацией о банках физической памяти

В результате на диске c:\ появится таблица стилей styles.css, задающая цвет и формат шрифта для тегов <body> (тело документа HTML), <table> (таблица), <th> (заголовок таблицы) и <td> (ячейка таблицы).

Для связывания HTML-файла с таблицей стилей можно вставить в раздел <head> данного файла следующий тег:

```
<link rel='stylesheet' href='c:\styles.css' type='text/ css' />
```

Данный тег добавляется в выходной файл c:\mem.html с помощью параметра -Head командлета ConvertTo-HTML:

```
PS C:\> Get-WmiObject Win32_PhysicalMemory | Select-Object BankLabel,
@{Name="Capacity, Mb"; Expression={$_.Capacity/1Mb}}, Description |
ConvertTo-HTML -Head "<link rel='stylesheet' href='c:\styles.css'
type='text/css' />" | Out-File C:\Mem.html
```

Откроем полученный файл c:\mem.html с помощью командлета `Invoke-Item`:

```
PS C:\> Invoke-Item c:\mem.html
```

Теперь таблица с результатами выглядит симпатичнее (рис. 16.2).

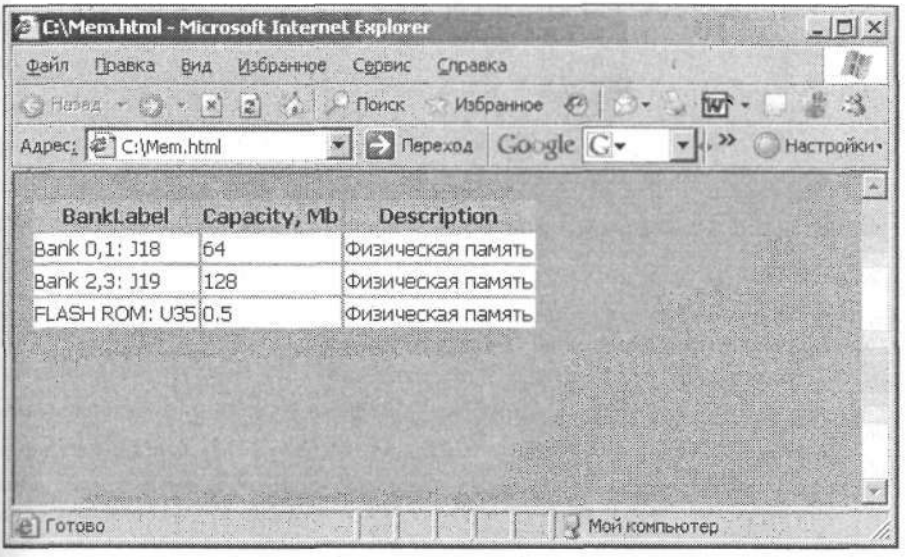


Рис. 16.2. HTML-страница с информацией о банках физической памяти (с дополнительным форматированием)

Мы будем пользоваться таблицей стилей c:\styles.css и в последующих разделах этой главы.

Получение информации о процессорах

Каждому процессору в WMI соответствует экземпляр класса `Win32_Processor`. Выведем сначала все свойства этого объекта:

```
PS C:\> Get-WmiObject Win32_Processor | Format-List *

__GENUS                   : 2
__CLASS                   : Win32_Processor
__SUPERCLASS              : CIM_Processor
__DYNASTY                 : CIM_ManagedSystemElement
__RELPATH                 : Win32_Processor.DeviceID="CPU0"
```

```
__PROPERTY_COUNT          : 44
__DERIVATION              : {CIM_Processor, CIM_LogicalDevice,
                            CIM_LogicalElement,
                            CIM_ManagedSystemElement}
__SERVER                  : POPOV
__NAMESPACE               : root\cimv2
__PATH                    : \\POPOV\root\cimv2:Win32_Processor.DeviceID
                            ="CPU0"
AddressWidth              : 32
Architecture              : 0
Availability              : 3
Caption                   : x86 Family 6 Model 8 Stepping 6
ConfigManagerErrorCode    :
ConfigManagerUserConfig   :
CpuStatus                 : 1
CreationClassName         : Win32_Processor
CurrentClockSpeed         : 846
CurrentVoltage            : 18
DataWidth                 : 32
Description               : x86 Family 6 Model 8 Stepping 6
DeviceID                  : CPU0
ErrorCleared              :
ErrorDescription          :
ExtClock                  : 100
Family                    : 17
InstallDate               :
L2CacheSize               : 256
L2CacheSpeed              : 846
LastErrorCode             :
Level                     : 6
LoadPercentage            : 1
Manufacturer              : GenuineIntel
MaxClockSpeed             : 846
Name                      : Процессор Intel Pentium III
OtherFamilyDescription    :
PNPDeviceID               :
PowerManagementCapabilities :
```

```
PowerManagementSupported    : False
ProcessorId                 : 0387F9FF00000686
ProcessorType               : 3
Revision                    : 2054
Role                        : CPU
SocketDesignation           : J1
Status                      : OK
StatusInfo                  : 3
Stepping                    : 6
SystemCreationClassName     : Win32_ComputerSystem
SystemName                  : POPOV
UniqueId                    :
UpgradeMethod               : 3
Version                     : Модель 8, Выпуск 6
VoltageCaps                 :
```

Оставим в отчете наиболее важные свойства:

```
PS C:\> Get-WmiObject Win32_Processor | Select-Object DeviceID, Name,
Manufacturer, Description, CurrentClockSpeed, MaxClockSpeed, L2CacheSize,
L2CacheSpeed

DeviceID          : CPU0
Name              : Процессор Intel Pentium III
Manufacturer      : GenuineIntel
Description       : x86 Family 6 Model 8 Stepping 6
CurrentClockSpeed : 846
MaxClockSpeed     : 846
L2CacheSize       : 256
L2CacheSpeed      : 846
```

Преобразуем полученный отчет о процессорах в формат HTML с использованием таблицы стилей c:\style.css, созданной в предыдущем разделе:

```
PS C:\> Get-WmiObject Win32_Processor | Select-Object DeviceID, Name,
Manufacturer, Description, CurrentClockSpeed, MaxClockSpeed, L2CacheSize,
L2CacheSpeed | ConvertTo-HTML -Head "<link rel='stylesheet'
href='c:\styles.css' type='text/css' />" | Out-File c:\proc.html
```

Откроем выходной файл c:\proc.html в веб-браузере с помощью командлета `Invoke-Item` (рис. 16.3):

```
PS C:\> Invoke-Item c:\proc.html
```

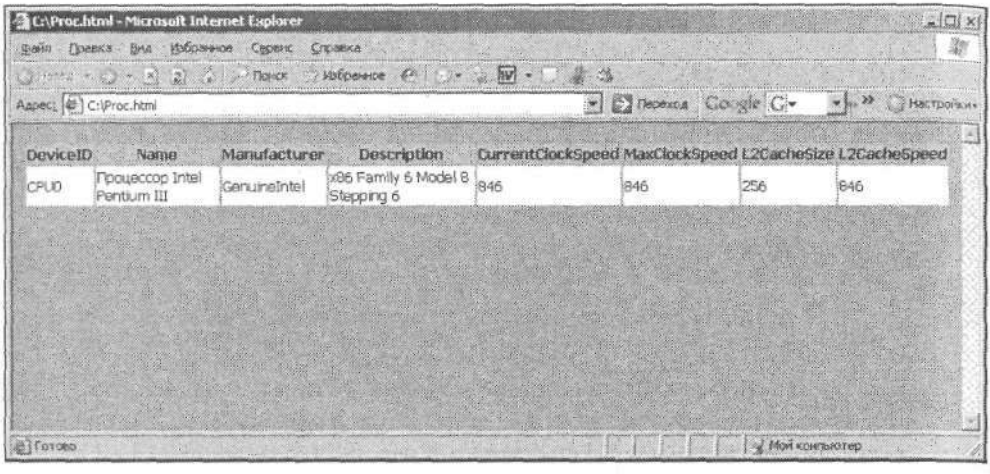


Рис. 16.3. HTML-страница с информацией о процессорах

Получение списка устройств Plug-and-Play

Устройствам Plug-and-Play, установленным в системе, соответствуют экземпляры класса `Win32_PnPEntity`. Подобных устройств в любой системе зарегистрировано будет много, поэтому мы сразу с помощью командлета `Select-Object` ограничим количество извлекаемых объектов:

```
PS C:\> Get-WmiObject Win32_PnPEntity | Select-Object -First 3 |
Format-List *

__GENUS                   : 2
__CLASS                   : Win32_PnPEntity
__SUPERCLASS              : CIM_LogicalDevice
__DYNASTY                 : CIM_ManagedSystemElement
__RELPATH                 : Win32_PnPEntity.DeviceID=
                            "ROOT\\ACPI_HAL\\0000"
__PROPERTY_COUNT          : 22
__DERIVATION              : {CIM_LogicalDevice, CIM_LogicalElement,
                            CIM_ManagedSystemElement}
__SERVER                  : POPOV
__NAMESPACE               : root\cimv2
__PATH                    : \\POPOV\root\cimv2:Win32_PnPEntity.
                            DeviceID="ROOT\\ACPI_HAL\\0000"
Availability              :
Caption                   : Компьютер с ACPI
```

```
ClassGuid                    : {4D36E966-E325-11CE-BFC1-08002BE10318}
ConfigManagerErrorCode       : 0
ConfigManagerUserConfig      : False
CreationClassName            : Win32_PnPEntity
Description                  : Компьютер с ACPI
DeviceID                     : ROOT\ACPI_HAL\0000
ErrorCleared                 :
ErrorDescription             :
InstallDate                  :
LastErrorCode                :
Manufacturer                 : (Стандартные компьютеры)
Name                         : Компьютер с ACPI
PNPDeviceID                  : ROOT\ACPI_HAL\0000
PowerManagementCapabilities  :
PowerManagementSupported     :
Service                      : \Driver\ACPI_HAL
Status                       : OK
StatusInfo                   :
SystemCreationClassName      : Win32_ComputerSystem
SystemName                   : POPOV
. . .
```

Оставим в отчете наиболее важные свойства:

```
PS C:\> Get-WmiObject Win32_PnPEntity | Select-Object Name, Manufacturer,
Status, PNPDeviceID -First 5 | Format-List *

Name         : Компьютер с ACPI
Manufacturer : (Стандартные компьютеры)
Status       : OK
PNPDeviceID  : ROOT\ACPI_HAL\0000

Name         : Microsoft ACPI-совместимая система
Manufacturer : Microsoft
Status       : OK
PNPDeviceID  : ACPI_HAL\PNP0C08\0

Name         : Шина PCI
Manufacturer : (стандартные системные устройства)
Status       : OK
```

```
PNPDeviceID  : ACPI\PNP0A03\2&DABA3FF&0

Name         : Intel 82443BX Pentium® II CPU - PCI мост
Manufacturer : Intel
Status       : OK
PNPDeviceID  : PCI\VEN_8086&DEV_7190&SUBSYS_00000000&REV_03\3&61AAA0
               1&0&00

Name         : Intel 82443BX Pentium® II CPU - AGP контроллер
Manufacturer : Intel
Status       : OK
PNPDeviceID  : PCI\VEN_8086&DEV_7191&SUBSYS_00000000&REV_03\3&61AAA0
               1&0&08
```

Преобразуем полученный отчет об устройствах Plug-and-Play в формат HTML с использованием таблицы стилей c:\style.css из предыдущих разделов:

```
PS C:\> Get-WmiObject Win32_PnPEntity | Select-Object Name, Manufacturer,
Status, PNPDeviceID | ConvertTo-HTML -Head "<link rel='stylesheet'
href='c:\styles.css' type='text/css' />" | Out-File c:\PnP.html
```

Откроем выходной файл c:\PnP.html в веб-браузере с помощью командлета `Invoke-Item` (рис. 16.4):

```
PS C:\> Invoke-Item c:\PnP.html
```

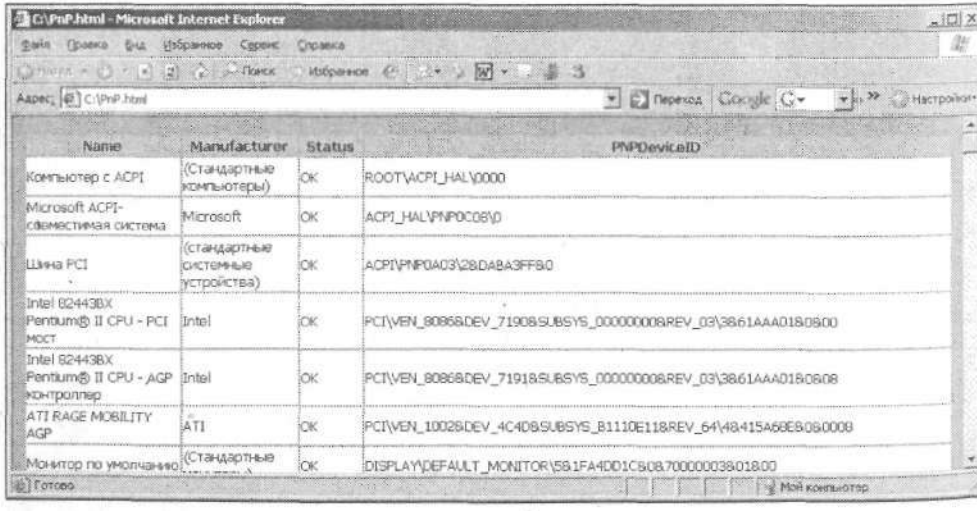


Рис. 16.4. HTML-страница с информацией об устройствах Plug-and-Play

Получение информации о звуковой карте

Доступ к настройкам звуковой карты, установленной в компьютере, можно получить с помощью экземпляра класса `Win32_SoundDevice`. Выведем некоторые наиболее важные свойства этого объекта:

```
PS C:\> Get-WmiObject Win32_SoundDevice | Format-List Name, Manufacturer,
Status, PNPDeviceID

Name         : Realtek AC'97 Audio
Manufacturer : Realtek
Status       : OK
PNPDeviceID  : PCI\VEN_8086&DEV_24D5&SUBSYS_02108086&REV_02\
               3&267A616A&0&FD
```

Как и в рассмотренных выше примерах, преобразуем выходную информацию в формат HTML и откроем полученный отчет в веб-браузере (рис. 16.5):

```
PS C:\> Get-WmiObject Win32_SoundDevice | Select-Object Name,
Manufacturer, Status, PNPDeviceID | ConvertTo-HTML -Head
"<link rel='stylesheet' href='c:\styles.css' type='text/css'  />" |
Out-File C:\Sound.html
PS C:\> Invoke-Item C:\Sound.html
```

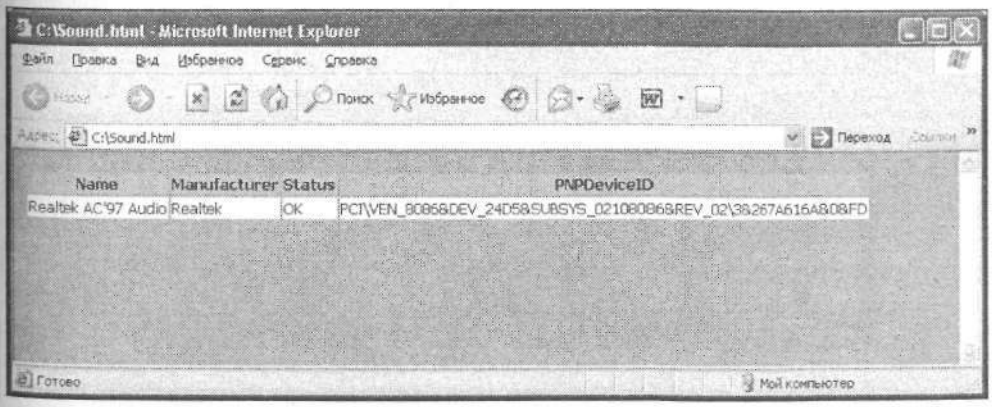


Рис. 16.5. HTML-страница с информацией о звуковой карте

Получение информации о видеокарте

Получить информацию о параметрах видеокарты, установленной в компьютере, можно с помощью объекта `Win32_VideoController`. Посмотрим сначала на все свойства этого объекта:

```
PS C:\> Get-WmiObject Win32_VideoController | Format-List *

__GENUS                    : 2
__CLASS                    : Win32_VideoController
__SUPERCLASS               : CIM_PCVideoController
__DYNASTY                  : CIM_ManagedSystemElement
__RELPATH                  : Win32_VideoController.DeviceID="Video
                             Controller1"
__PROPERTY_COUNT           : 59
__DERIVATION               : {CIM_PCVideoController,
                             CIM_VideoController, CIM_Controller,
                             CIM_LogicalDevice...}
__SERVER                   : 404-POPOV
__NAMESPACE                : root\cimv2
__PATH                     : \\404-POPOV\root\cimv2:
                             Win32_VideoController.DeviceID=
                             "VideoController1"
AcceleratorCapabilities    :
AdapterCompatibility       : Intel Corporation
AdapterDACType             : Internal
AdapterRAM                 : 100663296
Availability               : 3
CapabilityDescriptions     :
Caption                    : Intel(R) 82865G Graphics Controller
ColorTableEntries          :
ConfigManagerErrorCode     : 0
ConfigManagerUserConfig    : False
CreationClassName          : Win32_VideoController
CurrentBitsPerPixel        : 32
CurrentHorizontalResolution : 1280
CurrentNumberOfColors      : 4294967296
CurrentNumberOfColumns     : 0
CurrentNumberOfRows        : 0
```

```
CurrentRefreshRate          : 60
CurrentScanMode             : 4
CurrentVerticalResolution   : 1024
Description                 : Intel(R) 82865G Graphics Controller
DeviceID                    : VideoController1
DeviceSpecificPens          : 4294967295
DitherType                  :
DriverDate                  : 20050719073422.000000-000
DriverVersion               : 6.14.10.4363
ErrorCleared                :
ErrorDescription            :
ICMIntent                   :
ICMMethod                   :
InfFilename                 : oem0.inf
InfSection                  : i865G
InstallDate                 :
InstalledDisplayDrivers     : ialmrnt5.dll
LastErrorCode               :
MaxMemorySupported          :
MaxNumberControlled         :
MaxRefreshRate              : 75
MinRefreshRate              : 56
Monochrome                  : False
Name                        : Intel(R) 82865G Graphics Controller
NumberOfColorPlanes         : 1
NumberOfVideoPages          :
PNPDeviceID                 : PCI\VEN_8086&DEV_2572&SUBSYS_485A8086
                              &REV_02\3&267A616A&0&10
PowerManagementCapabilities :
PowerManagementSupported    :
ProtocolSupported           :
ReservedSystemPaletteEntries :
SpecificationVersion        :
Status                      : OK
StatusInfo                  :
SystemCreationClassName     : Win32_ComputerSystem
SystemName                  : 404-POPOV
SystemPaletteEntries        :
TimeOfLastReset             :
```

```
VideoArchitecture          : 5
VideoMemoryType            : 2
VideoMode                  :
VideoModeDescription       : 1280 x 1024 x 4294967296 цв.
VideoProcessor             : Intel(R) 82865G Graphics Controller
```

Выделим наиболее существенные свойства:

```
PS C:\> Get-WmiObject Win32_VideoController | Format-List Name,
AdapterDACType, Status, AdapterRAM, VideoModeDescription, MaxRefreshRate,
MinRefreshRate, DriverVersion, InstalledDisplayDrivers

Name                       : Intel(R) 82865G Graphics Controller
AdapterDACType             : Internal
Status                     : OK
AdapterRAM                 : 100663296
VideoModeDescription       : 1280 x 1024 x 4294967296 цв.
MaxRefreshRate             : 75
MinRefreshRate             : 56
DriverVersion              : 6.14.10.4363
InstalledDisplayDrivers    : ialmrnt5.dll
```

Преобразуем полученную информацию в формат HTML, включив в него строку для форматирования с использованием файла c:\styles.css из предыдущих разделов:

```
PS C:\> Get-WmiObject Win32_VideoController | Select-Object Name,
AdapterDACType, Status, AdapterRAM, VideoModeDescription, MaxRefreshRate,
MinRefreshRate, DriverVersion, InstalledDisplayDrivers | ConvertTo-HTML
-Head "<link rel='stylesheet' href='c:\styles.css' type='text/css' />" |
Out-File c:\video.html
```

Откроем полученный файл в веб-браузере (рис. 16.6):

```
PS C:\> Invoke-Item c:\video.html
```

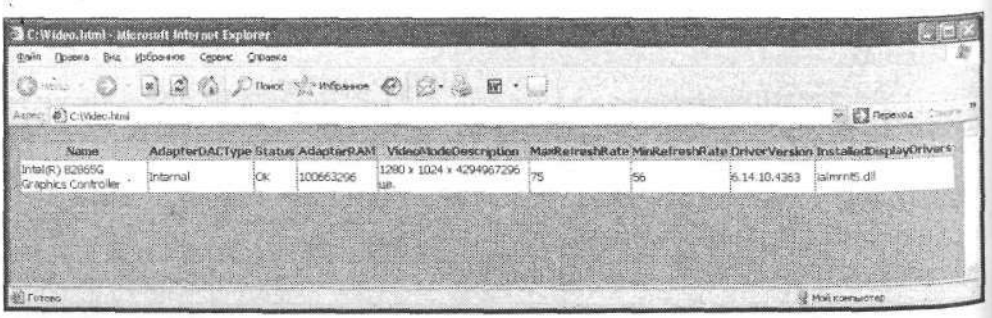


Рис. 16.6. HTML-страница с информацией о видеокарте

Получение информации о сетевых адаптерах

Получить информацию о сетевых адаптерах, зарегистрированных в системе, можно с помощью экземпляров класса `Win32_NetworkAdapter`. Выведем некоторые наиболее информативные свойства этих объектов:

```
PS C:\> Get-WmiObject Win32_NetworkAdapter | Format-List Caption,
Manufacturer, Installed, MACAddress, ServiceName

Caption      : [00000001] RAS асинхронный адаптер
Manufacturer :
Installed    : True
MACAddress   :
ServiceName  :

Caption      : [00000002] Минипорт WAN (L2TP)
Manufacturer : Microsoft
Installed    : True
MACAddress   :
ServiceName  : Rasl2tp

Caption      : [00000003] Минипорт WAN (PPTP)
Manufacturer : Microsoft
Installed    : True
MACAddress   : 50:50:54:50:30:30
ServiceName  : PptpMiniport

Caption      : [00000004] Минипорт WAN (PPPoE)
Manufacturer : Microsoft
Installed    : True
MACAddress   : 33:50:6F:45:30:30
ServiceName  : RasPppoe

Caption      : [00000005] Прямой параллельный порт
Manufacturer : Microsoft
Installed    : True
MACAddress   :
ServiceName  : Raspti

Caption      : [00000006] Минипорт WAN (IP)
Manufacturer : Microsoft
Installed    : True
MACAddress   :
ServiceName  : NdisWan
```

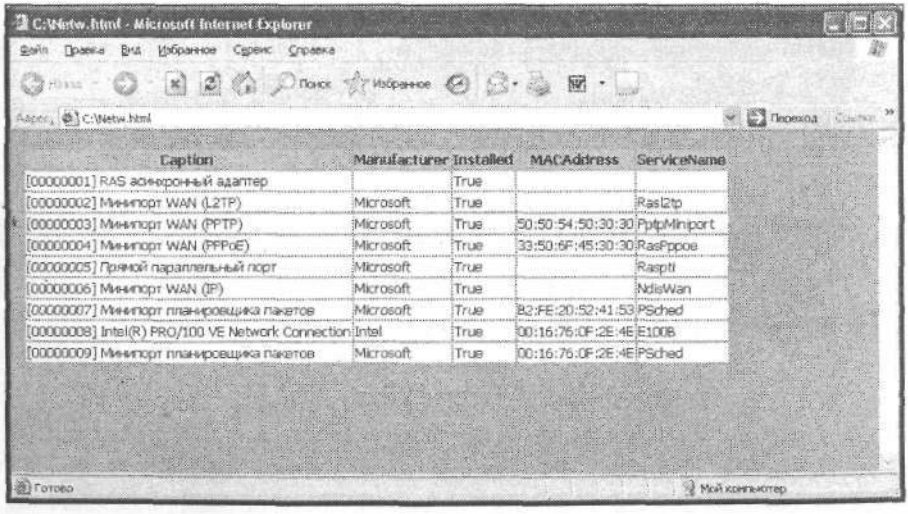
```
Caption      : [00000007] Минипорт планировщика пакетов
Manufacturer : Microsoft
Installed    : True
MACAddress   : 6A:91:20:52:41:53
ServiceName  : PSched

Caption      : [00000008] Intel(R) PRO/100 VE Network Connection
Manufacturer : Intel
Installed    : True
MACAddress   : 00:16:76:0F:2E:4E
ServiceName  : E100B

Caption      : [00000009] Минипорт планировщика пакетов
Manufacturer : Microsoft
Installed    : True
MACAddress   : 00:16:76:0F:2E:4E
ServiceName  : PSched
```

Преобразуем полученную информацию в формат HTML с использованием таблицы стилей c:\styles.css из предыдущих разделов:

```
PS C:\> Get-WmiObject Win32_NetworkAdapter | Select-Object Caption,
Manufacturer, Installed, MACAddress, ServiceName | ConvertTo-HTML -Head
"<link rel='stylesheet' href='c:\styles.css' type='text/css' />" |
Out-File c:\netw.html
```

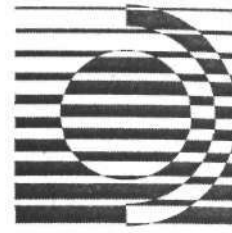
Откроем полученный файл в веб-браузере (рис. 16.7):

```
PS C:\> Invoke-Item c:\netw.html
```



Caption	Manufacturer	Installed	MACAddress	ServiceName
[00000001] RAS асинхронный адаптер		True		
[00000002] Минипорт WAN (L2TP)	Microsoft	True		Rasl2tp
[00000003] Минипорт WAN (PPTP)	Microsoft	True	50:50:54:50:30:30	PptpMiniport
[00000004] Минипорт WAN (PPPoE)	Microsoft	True	33:50:6F:45:30:30	RasPppoe
[00000005] Прямой параллельный порт	Microsoft	True		Raspti
[00000006] Минипорт WAN (IP)	Microsoft	True		NdisWan
[00000007] Минипорт планировщика пакетов	Microsoft	True	82:FE:20:52:41:53	PSched
[00000008] Intel(R) PRO/100 VE Network Connection	Intel	True	00:16:76:0F:2E:4E	E100B
[00000009] Минипорт планировщика пакетов	Microsoft	True	00:16:76:0F:2E:4E	PSched

Рис. 16.7. HTML-страница с информацией о сетевых адаптерах



Глава 17

Настройка сетевых параметров. Работа с электронной почтой

Любому администратору операционной системы наверняка неоднократно приходилось настраивать сетевые параметры рабочих станций, причем в сетях, где компьютеры меняются довольно часто, этим приходится заниматься постоянно. В данной главе мы рассмотрим примеры выполнения некоторых действий по настройке сетевых параметров из оболочки PowerShell, а также научимся автоматически формировать и отсылать сообщения по электронной почте из сценариев PowerShell.

Получение и настройка сетевых параметров

В современных системах большая часть задач по настройке сетевых параметров связана с протоколом TCP/IP, являющимся наиболее часто используемым сетевым протоколом. Доступ к настройкам TCP/IP можно получить с помощью графического интерфейса Windows, а также нескольких утилит командной строки (ipconfig.exe, netsh.exe, arp.exe и т. п.). Рассмотрим варианты решения некоторых подобных задач в оболочке PowerShell при помощи службы WMI (напомним, что классы WMI позволяют работать как с локальным, так и с удаленными компьютерами). Нам понадобятся WMI-классы `Win32_NetworkAdapter` и `Win32_NetworkAdapterConfiguration`, свойства и методы которых позволяют просматривать и изменять различные параметры и настройки сетевых адаптеров и протоколов. Экземплярами класса `Win32_NetworkAdapter`, которые соответствуют сетевым адаптерам, зарегистрированным в системе, мы уже пользовались в *главе 16*, когда рассматривали команды для получения сводной информации о сетевых адаптерах.

Замечание

Подсистема WMI может возвращать несколько экземпляров объектов `Win32_NetworkAdapter` и `Win32_NetworkAdapterConfiguration`, даже если в компьютере физически установлен только один сетевой адаптер. Это связано с тем, что некоторые службы и программные компоненты (RAS, PPTP, QoS и т. д.) создают и регистрируют в системе свои собственные виртуальные сетевые адаптеры.

С помощью экземпляров класса `Win32_NetworkAdapterConfiguration` можно получить подробную информацию о сетевых настройках соответствующего адаптера, включая параметры протокола TCP/IP. В классе `Win32_NetworkAdapterConfiguration` определено более 60 свойств и более 40 методов, позволяющих извлекать и изменять сетевые настройки.

Перейдем к рассмотрению примеров.

Получение списка IP-адресов компьютера

Для формирования списка IP-адресов локального компьютера мы сначала создадим коллекцию экземпляров класса `Win32_NetworkAdapterConfiguration` и сохраним ее в переменной `$NicConfig`:

```
PS C:\> $NicConfig=Get-WmiObject Win32_NetworkAdapterConfiguration
```

Значение IP-адреса хранится в свойстве `IPAddress` объектов `Win32_NetworkAdapterConfiguration`; воспользуемся командлетом `Select-Object` для выделения данного свойства:

```
PS C:\> $NicConfig | Select-Object IPAddress

IPAddress
---------

{10.169.7.26}
```

Пустые строки, возвращенные конвейером, показывают, что для некоторых объектов свойство `IPAddress` не инициализировано, так как для соответствующих сетевых адаптеров протокол TCP/IP не включен. Для исключения подобных объектов из списка можно предварительно выполнить фильтрацию в конвейере по значению свойства `IPEnabled`:

```
PS C:\> $NicConfig | Where-Object {$_.IPEnabled} | Select-Object IPAddress

IPAddress
---------
{10.169.7.26}
```

Обратим внимание, что значения IP-адресов выводятся в фигурных скобках (у одного сетевого адаптера может быть несколько IP-адресов). Это связано с тем, что свойство `IPAddress` является не строкой, а массивом строк:

```
PS C:\> $NicConfig | Get-Member


   TypeName: System.Management.ManagementObject#root\cimv2\
Win32_NetworkAdapterConfiguration

Name                        MemberType   Definition
----                        ----------   ----------
DisableIPSec                Method       System.Management.Manag...
. . .
IPAddress                   Property     System.String[] IPAddre...
. . .
```

Для раскрытия массива `IPAddress` можно воспользоваться параметром `-ExpandProperty` командлета `Select-Object`:

```
PS C:\> $NicConfig | Where-Object {$_.IPEnabled} |
>> Select-Object -ExpandProperty IPAddress
>>
10.169.7.26
```

Вывод параметров протокола TCP/IP

Для отображения параметров протокола TCP/IP для сетевых адаптеров нужно, как и в предыдущем разделе, оставить в коллекции экземпляров класса `Win32_NetworkAdapterConfiguration` только те объекты, для которых значение свойства `IPEnabled` равно `$True`:

```
PS C:\> $NicConfig=Get-WmiObject Win32_NetworkAdapterConfiguration
PS C:\> $NicConfig | Where-Object {$_.IPEnabled}


DHCPEnabled      : True
IPAddress        : {10.169.7.26}
DefaultIPGateway : {10.169.7.1}
DNSDomain        :
ServiceName      : E100B
Description      : Intel(R) PRO/100 VE Network Connection - Минипорт
                   планировщика пакетов
Index            : 8
```

Как видим, по умолчанию для объектов Win32_NetworkAdapterConfiguration на экране отображается только небольшая часть сведений о конфигурации сетевого адаптера. Для отображения всех свойств можно воспользоваться командлетом Select-Object:

```
PS C:\> $NicConfig | Where-Object {$_.IPEnabled} | Select-Object *

DHCPLeaseExpires             : 20080605135529.000000+240
Index                        : 8
Description                  : Intel(R) PRO/100 VE Network Connection -
                               Минипорт планировщика пакетов
DHCPEnabled                  : True
DHCPLeaseObtained            : 20080604175529.000000+240
DHCPServer                   : 10.169.7.3
DNSDomain                    :
DNSDomainSuffixSearchOrder   :
DNSEnabledForWINSResolution  : False
DNSHostName                  : popov
DNSServerSearchOrder         :
DomainDNSRegistrationEnabled : False
FullDNSRegistrationEnabled   : True
IPAddress                    : {10.169.7.26}
IPConnectionMetric           : 20
IPEnabled                    : True
IPFilterSecurityEnabled      : False
WINSEnableLMHostsLookup      : True
WINSHostLookupFile           :
WINSPrimaryServer            : 10.169.1.15
WINSScopeID                  :
WINSSecondaryServer          : 10.169.1.69
__GENUS                      : 2
__CLASS                      : Win32_NetworkAdapterConfiguration
__SUPERCLASS                 : CIM_Setting
__DYNASTY                    : CIM_Setting
__RELPATH                    : Win32_NetworkAdapterConfiguration.Index=8

__PROPERTY_COUNT             : 60
__DERIVATION                 : {CIM_Setting}
__SERVER                     : POPOV
```

```
__NAMESPACE                    : root\cimv2
__PATH                         : \\POPOV\root\cimv2:
                                 Win32_NetworkAdapterConfiguration.Index=8
ArpAlwaysSourceRoute           :
ArpUseEtherSNAP                :
Caption                        : [00000008] Intel(R) PRO/100 VE Network
                                 Connection
DatabasePath                   : %SystemRoot%\System32\drivers\etc
DeadGWDetectEnabled            :
DefaultIPGateway               : {10.169.7.1}
DefaultTOS                     :
DefaultTTL                     :
ForwardBufferMemory            :
GatewayCostMetric              : {20}
IGMPLevel                      :
IPPortSecurityEnabled          :
IPSecPermitIPProtocols         : {0}
IPSecPermitTCPPorts            : {0}
IPSecPermitUDPPorts            : {0}
IPSubnet                       : {255.255.255.0}
IPUseZeroBroadcast             :
IPXAddress                     :
IPXEnabled                     : False
IPXFrameType                   :
IPXMediaType                   :
IPXNetworkNumber               :
IPXVirtualNetNumber            :
KeepAliveInterval              :
KeepAliveTime                  :
MACAddress                     : 00:16:76:0F:2E:4E
MTU                            :
NumForwardPackets              :
PMTUBHDetectEnabled            :
PMTUDiscoveryEnabled           :
ServiceName                    : E100B
SettingID                      : {5A38EE0D-89A5-4207-BD82-9BDF5A988E45}
TcpipNetbiosOptions            : 0
TcpMaxConnectRetransmissions   :
```

```
TcpMaxDataRetransmissions     :
TcpNumConnections             :
TcpUseRFC1122UrgentPointer    :
TcpWindowSize                 :
```

Для исключения служебных параметров WMI, начинающихся с символа подчеркивания, можно в качестве значения параметра `-Property` в командлете `Select-Object` указать шаблон `[a-z]*`. Если некоторые свойства не представляют интереса (например, параметры протокола IPX), то их можно указать в качестве значения параметра `-ExcludeProperty`. Например, сле-дующая команда выведет информацию о сетевых адаптерах за исключением служебных свойств WMI и свойств, связанных с протоколом IPX:

```
PS C:\> $NicConfig | Where-Object {$_.IPEnabled} |
>> Select-Object -Property [a-z]* -ExcludeProperty ipx*

DHCPLeaseExpires              : 20080605135529.000000+240
Index                         : 8
Description                   : Intel(R) PRO/100 VE Network Connection -
                                Минипорт планировщика пакетов
DHCPEnabled                   : True
DHCPLeaseObtained             : 20080604175529.000000+240
DHCPServer                    : 10.169.7.3
DNSDomain                     :
DNSDomainSuffixSearchOrder    :
DNSEnabledForWINSResolution   : False
DNSHostName                   : popov
DNSServerSearchOrder          :
DomainDNSRegistrationEnabled  : False
FullDNSRegistrationEnabled    : True
IPAddress                     : {10.169.7.26}
IPConnectionMetric            : 20
IPEnabled                     : True
IPFilterSecurityEnabled       : False
WINSEnableLMHostsLookup       : True
WINSHostLookupFile            :
WINSPrimaryServer             : 10.169.1.15
WINSScopeID                   :
WINSSecondaryServer           : 10.169.1.69
ArpAlwaysSourceRoute          :
```

```
ArpUseEtherSNAP                :
Caption                        : [00000008] Intel(R) PRO/100 VE Network
                                 Connection
DatabasePath                   : %SystemRoot%\System32\drivers\etc
DeadGWDetectEnabled            :
DefaultIPGateway               : {10.169.7.1}
DefaultTOS                     :
DefaultTTL                     :
ForwardBufferMemory            :
GatewayCostMetric              : {20}
IGMPLevel                      :
IPPortSecurityEnabled          :
IPSecPermitIPProtocols         : {0}
IPSecPermitTCPPorts            : {0}
IPSecPermitUDPPorts            : {0}
IPSubnet                       : {255.255.255.0}
IPUseZeroBroadcast             :
KeepAliveInterval              :
KeepAliveTime                  :
MACAddress                     : 00:16:76:0F:2E:4E
MTU                            :
NumForwardPackets              :
PMTUBHDetectEnabled            :
PMTUDiscoveryEnabled           :
ServiceName                    : E100B
SettingID                      : {5A38EE0D-89A5-4207-BD82-9BDF5A988E45}
TcpipNetbiosOptions            : 0
TcpMaxConnectRetransmissions   :
TcpMaxDataRetransmissions      :
TcpNumConnections              :
TcpUseRFC1122UrgentPointer     :
TcpWindowSize                  :
```

Основные параметры протокола TCP/IP можно вывести следующей командой:

```
PS C:\> $NicConfig | Where-Object {$_.IPEnabled} | Select-Object Index,
>> Description, DHCPEnabled, IPAddress, IPSubnet, DefaultIPGateway,
>> GatewayCostMetric, IPConnectionMetric, DNSDomain
>>

Index                : 8
```

```
Description          : Intel(R) PRO/100 VE Network Connection - Минипорт
                       планировщика пакетов
DHCPEnabled          : True
IPAddress            : {10.169.7.26}
IPSubnet             : {255.255.255.0}
DefaultIPGateway     : {10.169.7.1}
GatewayCostMetric    : {20}
IPConnectionMetric : 20
DNSDomain            :
```

Настройка DHCP

Одной из наиболее часто возникающих перед администратором сети задач является настройка на рабочих станциях параметров протокола DHCP (Dynamic Host Configuration Protocol, протокол динамической настройки сетевых узлов), применяемого для упрощения процесса управления клиентами TCP/IP в сети. С помощью этого протокола клиент может автоматически получать от DHCP-сервера IP-адреса, маски подсетей и другие параметры TCP/IP, что позволяет не настраивать вручную статический IP-адрес для каждой рабочей станции в сети.

При загрузке клиента DHCP он обращается к серверу DHCP с запросом на аренду IP-адреса. Сервер DHCP отвечает на запрос, выбирая доступный IP-адрес из области адресов, которой он управляет. После этого сервер выделяет клиенту выбранный адрес на определенный промежуток времени (по умолчанию это восемь дней), предоставляя клиенту связанную с этим адресом маску подсети и, в случае необходимости, дополнительную информацию (адрес шлюза по умолчанию, а также адреса серверов DNS и WINS). После получения аренды клиент должен периодически обновлять ее, запрашивая у сервера IP-адрес.

Рассмотрим варианты выполнения в оболочке PowerShell нескольких типичных действий по настройке параметров DHCP.

Определение адаптеров, поддерживающих DHCP

Для выделения сетевых адаптеров, поддерживающих протокол DHCP, нужно анализировать значение свойства `DHCPEnabled` у экземпляров класса `Win32_NetworkAdapterConfiguration`. Если значение данного свойства равно `$True`, то для соответствующего сетевого адаптера протокол DHCP включен:

```
PS C:\> $NicConfig=Get-WmiObject Win32_NetworkAdapterConfiguration
PS C:\> $NicConfig | Where-Object {$_.DHCPEnabled}
```

```
DHCPEnabled       : True
IPAddress         : {10.169.7.26}
DefaultIPGateway  : {10.169.7.1}
DNSDomain         :
ServiceName       : E100B
Description       : Intel(R) PRO/100 VE Network Connection - Минипорт
                    планировщика пакетов
Index             : 8
```

Нужные объекты можно выделить, и не обращаясь к командлету `Where-Object`, поставив условие фильтрации объектов в соответствующий WQL-запрос:

```
PS C:\> $NicDHCP=Get-WmiObject -Query 'select * from
Win32_NetworkAdapterConfiguration where DHCPEnabled=True'
PS C:\> $NicDHCP

DHCPEnabled       : True
IPAddress         : {10.169.7.26}
DefaultIPGateway  : {10.169.7.1}
DNSDomain         :
ServiceName       : E100B
Description       : Intel(R) PRO/100 VE Network Connection - Минипорт
                    планировщика пакетов
Index             : 8
```

Просмотр параметров протокола DHCP

Параметры протокола DHCP для определенного сетевого адаптера хранятся в свойствах соответствующего объекта `Win32_NetworkAdapterConfiguration`, название которых начинается с "DHCP" (табл. 17.1).

Таблица 17.1. *Свойства класса* `Win32_NetworkAdapterConfiguration`*, относящиеся к DHCP*

Свойство	Описание
`DHCPEnabled`	Показывает, разрешен ли протокол DHCP (это свойство логического типа)
`DHCPLeaseExpires`	Содержит дату окончания срока действия IP-адреса, предоставленного сервером DHCP
`DHCPLeaseObtained`	Указывает дату и время предоставления компьютеру IP-адреса сервером DHCP
`DHCPServer`	Содержит IP-адрес сервера DHCP

Вначале мы создадим коллекцию объектов, экземпляры которой соответствуют сетевым адаптерам с разрешенным протоколом DHCP:

```
PS C:\> $NicDHCP=Get-WmiObject -Query 'select * from
Win32_NetworkAdapterConfiguration where DHCPEnabled=True'
```

Выделить свойства, относящиеся к протоколу DHCP, можно с помощью параметра `-Property` командлета `Select-Object`:

```
PS C:\> $NicDHCP | Select-Object -Property DHCP*

Description          : Intel(R) PRO/100 VE Network Connection - Минипорт
                       планировщика пакетов
DHCPEnabled          : True
DHCPLeaseExpires     : 20080609041043.000000+240
DHCPLeaseObtained    : 20080608081043.000000+240
DHCPServer           : 10.169.7.3
Index                : 8
```

Включение/отключение поддержки DHCP на сетевых адаптерах

Для включения поддержки протокола DHCP на сетевом адаптере нужно вызвать метод `EnableDHCP` соответствующего экземпляра класса `Win32_NetworkAdapterConfiguration`. Этот метод, как и другие методы класса `Win32_NetworkAdapterConfiguration`, возвращает целочисленное значение, по которому можно определить, успешно ли он выполнен. Полный список значений, возвращаемых методами класса `Win32_NetworkAdapterConfiguration`, приведен в табл. 17.2.

Таблица 17.2. *Значения, возвращаемые методами класса* `Win32_NetworkAdapterConfiguration`

Значение	Описание
0	Метод завершен успешно, перезагрузка компьютера не требуется
1	Метод завершен успешно, требуется перезагрузка компьютера
64	На этой платформе метод не поддерживается
65	Неизвестная ошибка
66	Неправильная маска подсети
67	При обработке возвращенного экземпляра произошла ошибка
68	Неправильный параметр ввода
69	Указано более 5 шлюзов

Таблица 17.2 (окончание)

Значение	Описание
70	Недопустимый IP-адрес
71	Неправильный IP-адрес шлюза
72	При попытке доступа к реестру за запрошенной информацией произошла ошибка
73	Неправильное имя домена
74	Неправильное имя узла
75	Не указаны основной и дополнительный WINS-серверы
76	Неправильный файл
77	Неправильный системный путь
78	Не удалось скопировать файл
79	Недопустимый параметр безопасности
80	Не удается настроить службу TCP/IP
81	Не удается настроить службу DHCP
82	Не удалось обновить аренду DHCP
83	Не удалось освободить аренду DHCP
84	Протокол IP не разрешен на адаптере
85	Протокол IPX не разрешен на адаптере
86	Выход за допустимые границы номера кадра или сети
87	Недопустимый тип кадра
88	Недопустимый номер сети
89	Такой номер сети уже существует
90	Выход параметра за допустимые границы
91	Отказано в доступе
92	Недостаточно памяти
93	Объект уже существует
94	Путь, файл или объект не найден
95	Не удалось уведомить службу
96	Не удалось уведомить службу DNS
97	Интерфейс не может быть настроен
98	Не все аренды DHCP удалось освободить или обновить
100	DHCP не разрешено для этого адаптера

Если нужно разрешить DHCP на всех сетевых адаптерах, поддерживающих протокол IP, то можно выполнить следующую команду:

```
PS C:\> Get-WmiObject -Query 'select * from
Win32_NetworkAdapterConfiguration where DHCPEnabled=True' |
ForEach-Object -Process {$_.EnableDHCP()}


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Как видим, в результате выполнения метода `EnableDHCP` формируется объект, в свойстве `ReturnValue` которого содержится значение, возвращаемое данным методом.

Для назначения сетевому адаптеру статического IP-адреса и, как следствие, отключения DHCP на этом адаптере используется метод `EnableStatic`, в качестве параметров которого указываются устанавливаемый IP-адрес и маска подсети. Например, следующая команда назначает сетевому адаптеру с индексом 8 статический IP-адрес 10.169.7.26 с маской подсети 255.255.255.0:

```
PS C:\> (Get-WmiObject -Query 'select * from
Win32_NetworkAdapterConfiguration where IPEnabled=True and In-
dex=8').EnableStatic("10.169.7.26", "255.255.255.0")


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Отмена и обновление аренды DHCP

Объекты `Win32_NetworkAdapterConfiguration` имеют методы `ReleaseDHCPLease` и `RenewDHCPLease`, с помощью которых можно соответственно отменить и обновить аренду DHCP на соответствующих сетевых адаптерах. Например, следующая команда отменяет аренду DHCP на адаптере, для которого установлен адрес сервера DHCP 10.169.7.3:

```
PS C:\> Get-WmiObject -Query 'select * from
Win32_NetworkAdapterConfiguration where DHCPEnabled=True' |
Where-Object {$_.DHCPServer -contains "10.169.7.3"} |
ForEach-Object {$_.ReleaseDHCPLease()}


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Обновление аренды DHCP для того же адаптера производится аналогично:

```
PS C:\> Get-WmiObject -Query 'select * from
Win32_NetworkAdapterConfiguration where DHCPEnabled=True' |
Where-Object {$_.DHCPServer -contains "10.169.7.3"} |
ForEach-Object {$_.RenewDHCPLease()}


__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Если нужно отменить или обновить аренду DHCP для всех сетевых адаптеров, то можно воспользоваться методами `ReleaseDHCPLeaseAll` и `RenewDHCPLeaseAll`. При этом следует учитывать, что данные методы являются методами класса, а не отдельных экземпляров, поэтому вызывать их нужно специальным образом. Ссылку на класс WMI вместо ссылки на экземпляры этого класса можно получить путем вывода всех классов WMI (командлет `Get-WMIObject` с параметром `-List`) и выбора (с помощью командлета `Where-Object`) нужного класса по его имени. Следующий конвейер команд вызывает метод `ReleaseDHCPLeaseAll`:

```
PS C:\> (Get-WmiObject -List | Where-Object {$_.Name -eq
>> "Win32_NetworkAdapterConfiguration"}).RenewDHCPLeaseAll()
>>

__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Отправка сообщений по электронной почте

Предположим, что вы написали сценарий, который проверяет определенные параметры системы (например, объем свободного места на жестком диске сервера) и при достижении этими параметрами критических значений должен уведомить администратора. Часто удобнее всего бывает получать подобные уведомления в виде сообщений по электронной почте, поэтому возникает задача формирования и отправки такого сообщения в сценарии PowerShell. Для этого можно воспользоваться .NET-классом `System.Net.Mail.MailMessage`. Мы рассмотрим наиболее распространенный случай отправки сообщения по протоколу SMTP.

Сначала определим несколько переменных, в которых будут храниться атрибуты создаваемого сообщения. В переменную `$sender` запишем адрес отправителя:

```
PS C:\> $sender = "sender@somedomain.com"
```

В переменной `$recipient` сохраним адрес получателя:

```
PS C:\> $recipient = "recipient@somedomain.com"
```

Переменная `$server` будет содержать имя сервера SMTP:

```
PS C:\> $server = "SMTPServer"
```

Тему письма запишем в переменную `$subject`:

```
PS C:\> $subject = "Сообщение от PowerShell"
```

Тело письма сохраним в переменной `$body`:

```
PS C:\> $body = "Это сообщение создано сценарием PowerShell"
```

Теперь создадим экземпляр класса `System.Net.Mail.MailMessage`, указав в качестве параметров конструктора адрес отправителя, адрес получателя, тему и тело сообщения. Ссылку на созданный объект сохраним в переменной `$msg`:

```
PS C:\> $msg = New-Object System.Net.Mail.MailMessage $sender,
>> $recipient, $subject, $body
>>
```

В переменной `$client` сохраним ссылку на объект `System.Net.Mail.SmtpClient`, указав в качестве параметра конструктора имя сервера SMTP:

```
PS C:\> $client = New-Object System.Net.Mail.SmtpClient $server
```

Теперь нужно указать учетные данные (имя и пароль) пользователя, от имени которого будет отослано письмо. Эти данные должны быть записаны в свойство `Credentials` переменной `$client` в виде объекта `System.Net.NetworkCredential`. В качестве параметров конструктора объектов передаются имя пользователя ("popov-av") и пароль ("password"):

```
PS C:\> $client.Credentials=New-Object System.Net.NetworkCredential
"popov-av", "password"
```

Если сообщение нужно отправить с использованием учетных данных работающего в данный момент пользователя, то свойство `Credentials` можно заполнить следующим образом:

```
PS C:\> $client.Credentials=
>> [System.Net.CredentialCache]::DefaultNetworkCredentials
>>
```

Сообщение сформировано. Для его отправки достаточно вызвать метод `Send` объекта `System.Net.Mail.SmtpClient`, указав в качестве параметра ссылку

на соответствующий объект `System.Net.Mail.MailMessage` (в нашем случае это переменная `$msg`):

```
PS C:\> $client.Send($msg)
```

Сообщение отослано. Полностью сценарий SendMessage.ps1, формирующий и отсылающий сообщение по протоколу SMTP, приведен в листинге 17.1.

Листинг 17.1. Формирование и отсылка сообщения по протоколу SMTP (файл SendMessage.ps1)

```
##############################################################
# Имя: SendMessage.ps1
# Язык: PoSH 1.0
# Описание: Формирование и отсылка сообщения по протоколу SMTP
##############################################################
#Задаем адрес отправителя
$sender = "sender@somedomain.com"
#Задаем адрес получателя
$recipient = "recipient@somedomain.com"
#Задаем имя сервера
$server = "SMTPServer"
#Определяем заголовок сообщения
$subject = "Сообщение от PowerShell"
#Формируем тело сообщения
$body = "Это сообщение создано сценарием PowerShell"

#Создаем экземпляр класса System.Net.Mail.MailMessage, соответствующий
#сообщению
$msg = New-Object System.Net.Mail.MailMessage $sender, $recipient,
       $subject, $body

#Создаем экземпляр класса System.Net.Mail.SmtpClient, соответствующий
#клиенту SMTP для указанного сервера SMTP
$client = New-Object System.Net.Mail.SmtpClient $server

#Указываем имя и пароль для подключения к серверу SMTP
$client.Credentials = New-Object System.Net.NetworkCredential "popov-av",
                      "password"

#Отсылаем сообщение
$client.Send($msg)

### Конец сценария ########################
```

Глава 18

PowerShell, cmd.exe и VBScript: совместное использование

Напомним еще раз, что до появления оболочки PowerShell основными средствами автоматизации в операционной системе Windows были командная оболочка cmd.exe со своими пакетными (командными) файлами и сценарии (преимущественно на языке VBScript), выполняемые под управлением сервера сценариев Windows Script Host. Эти инструменты использовались много лет, и за эти годы разработано множество вариантов решения различных задач автоматизации, которые можно легко найти в Интернете или в специальной литературе и доработать под свои конкретные задачи.

И вот появляется новая оболочка командной строки PowerShell со своим специфическим языком, работающая на принципиально других базовых принципах. Естественно, у администратора операционной системы, который начинает разбираться с PowerShell, возникают вопросы. Можно ли в новой оболочке пользоваться уже имеющимися командными файлами и сценариями на языке VBScript? Какие при этом действуют ограничения? Какие особенности нужно учитывать при переводе имеющихся сценариев и командных файлов на язык PowerShell?

На самом деле во многих случаях PowerShell можно применять в качестве связующего звена между уже имеющимися решениями задач автоматизации, хотя иногда бывает эффективнее и проще для дальнейшего использования целиком переписать VBScript-сценарий или командный файл на язык PowerShell. В любом случае для грамотной работы желательно четко представлять себе отличия PowerShell от языков cmd.exe и VBScript, знать способы запуска в новой оболочке уже имеющихся командных файлов и сценариев, написанных на этих языках, а также уметь переводить имеющиеся сценарии на язык PowerShell. Рассмотрению данных вопросов и посвящена настоящая глава.

Сравнение языков PowerShell и cmd.exe

Давайте рассмотрим основные отличия языков PowerShell и cmd.exe, с которыми придется столкнуться при переходе на новую оболочку командной строки и при переводе командных файлов на PowerShell.

Различия в синтаксисе команд

Напомним, что в оболочке PowerShell определены псевдонимы для совместимости имен базовых команд с именами команд оболочки cmd.exe и с именами команд оболочек операционной системы Unix. Это позволяет пользователю, привыкшему работать в оболочке cmd.exe, вводить знакомые имена команд и утилит (`dir`, `sort`, `type` и т. д.), получая ожидаемый результат.

Естественно, точного соответствия параметров команд в двух различных оболочках быть не может. Нужно учитывать, что cmd.exe и PowerShell различаются по принципу реализации функциональности в командах. Командлеты PowerShell выполняют, как правило, одну определенную операцию и имеют небольшое количество стандартных параметров, однако могут легко соединяться друг с другом посредством конвейера, образуя "составные" команды. Оболочка cmd.exe, напротив, имеет небольшое количество стандартных команд, каждая из которых может выполнять различные действия в зависимости от указанной при ее запуске комбинации параметров (состав и значение параметров у разных команд различаются). Поэтому PowerShell-эквивалент для одной команды cmd.exe с разными параметрами может состоять из нескольких командлетов, соединенных в конвейер. Для иллюстрации данного факта в табл. 18.1 приведены примеры команд интерпретатора cmd.exe, часто используемых при работе с файловой системой, и их эквиваленты на языке PowerShell (с псевдонимами и без них).

Таблица 18.1. *Некоторые команды в cmd.exe и PowerShell*

Описание операции	Команда cmd.exe	Команда PowerShell (с псевдонимами)	Команда PowerShell
Получение списка файлов в текущем каталоге	`dir`	`dir`	`Get-ChildItem`
Получение списка файлов, имена которых соответствуют шаблону	`dir *.rtf`	`dir *.rtf`	`Get-ChildItem *.rtf`

Таблица 18.1 (окончание)

Описание операции	Команда cmd.exe	Команда PowerShell (с псевдонимами)	Команда PowerShell
Получение списка файлов в текущем каталоге и всех его подкаталогах	`dir /s`	`dir -rec`	`Get-ChildItem -Recurse`
Получение списка всех текстовых файлов в текущем каталоге и всех его подкаталогах	`dir /s *.txt`	`dir -rec -fi *.txt`	`Get-ChildItem -Recurse -Filter *.txt`
Получение списка файлов в текущем каталоге, упорядоченного по времени последнего изменения файлов	`dir /o:-d`	`dir | sort -desc LastWriteTime`	`Get-ChildItem | Sort-Object -Descending LastWriteTime`
Получение списка файлов в текущем каталоге в кратком формате (только имена файлов)	`dir /b`	`dir -name`	`Get-ChildItem -Name`
Получение списка подкаталогов текущего каталога	`dir /a:d`	`dir | ? {$_.PSIsContainer}`	`Get-ChildItem | Where-Object {$_.PSIsContainer}`
Установка текущего каталога	`cd c:\`	`cd c:\`	`Set-Location c:\`
Вывод содержимого текстового файла на экран	`type 1.txt`	`type 1.txt`	`Get-Content 1.txt`
Копирование файла	`copy c:\1.txt d:\`	`copy c:\1.txt d:\`	`Copy-Item c:\1.txt d:\`
Конкатенация нескольких файлов	`copy 1.txt +2.txt 3.txt`	`type 1.txt,2.txt > .\3.txt`	`Get-Content 1.txt,2.txt > .\3.txt`
Удаление файла	`del 1.txt`	`del 1.txt`	`Remove-Item 1.txt`
Удаление всех текстовых файлов в текущем каталоге	`del *.txt`	`del *.txt`	`Remove-Item *.txt`
Удаление всех текстовых файлов в текущем каталоге и его подкаталогов	`del /s *.txt`	`del -rec *.txt`	`Remove-Item -Recurse *.txt`

При работе в cmd.exe часто используются операторы перенаправления ввода/вывода. Оболочка PowerShell поддерживает конвейеризацию команд (это основной механизм PowerShell), а также все операторы перенаправления вывода (>, >>, 2>, 2>&1), имеющие тот же смысл, что и в оболочке cmd.exe. Оператор перенаправления ввода < в PowerShell не поддерживается, вместо него следует использовать командлет `Get-Content` (псевдоним `type`).

Работа с переменными

В оболочке cmd.exe значения переменным присваиваются с помощью команды `set`, например:

```
C:\> set a=1
```

Для получения значения переменной ее имя нужно заключить в знаки процента `%`:

```
C:\> echo a=%a%
a=1
```

В PowerShell перед именами переменных должен стоять знак `$`. Для присвоения значения переменной не требуется специальный оператор — достаточно простого присваивания, например:

```
PS C:\> $a=1
```

Для получения значения переменной также не нужно указывать никаких дополнительных символов:

```
PS C:\> echo a=$a
a=1
```

Все переменные, с которыми работает оболочка cmd.exe, являются переменными среды Windows. Оболочка PowerShell поддерживает разные типы переменных; доступ к переменным среды осуществляется через виртуальный диск `Env:`. Например, для получения в PowerShell значения переменной среды OS нужно выполнить следующую команду:

```
PS C:\> $env:OS
Windows_NT
```

Для проведения арифметических вычислений в оболочке cmd.exe применяется команда `set` с ключом `/a`, например:

```
C:\> set /a n=1+3
4
C:\> echo %n%
4
```

В PowerShell для проведения вычислений нет необходимости использовать какой-либо специальный синтаксис — достаточно просто написать нужное выражение, например:

```
PS C:\> $n=1+2
PS C:\> $n
3
```

Так как PowerShell базируется на объектах .NET, в этой оболочке можно выполнять любые математические операции, доступные в языках типа C# или Visual Basic, в том числе вычислять значения математических функций с плавающей запятой, например:

```
PS C:\> $n=[math]::sqrt(9)+[math]::sin([math]::pi/4)
PS C:\> $n
3.70710678118655
```

В оболочке cmd.exe можно выполнять некоторые преобразования над переменными, как над символьными строками. Например, можно заменять символы в значении переменной:

```
C:\> set a=123456
C:\> set b=%a:23=99%
C:\> echo %b%
199456
```

В данном примере в переменную `a` записывается строка "123456", а в переменную `b` заносится строка "199456" — результат замены подстроки "23" на подстроку "99".

В PowerShell подобные преобразования выполняются с помощью выражений и операторов. Последний рассмотренный пример на языке PowerShell будет иметь следующий вид:

```
PS C:\> $a="123456"
PS C:\> $b=$a -replace "23", "99"
PS C:\> $b
199456
```

Использование циклов

В оболочке cmd.exe для организации циклов используются различные вариации команды `for`, и некоторые из них имеют довольно сложный и запутанный синтаксис. В PowerShell можно получить ту же функциональность с помощью конвейера из нескольких команд. В табл. 18.2 приведены несколько примеров циклов, организованных с помощью команды `for`, и их эквиваленты на языке PowerShell (с псевдонимами и без них).

Таблица 18.2. Примеры циклов в cmd.exe и PowerShell

Описание действия	Цикл в cmd.exe	Цикл в PowerShell (с псевдонимами)
Перебор файлов в текущем каталоге	`for %f in (*) do echo %f`	`dir | ? {!$_.PSIsContainer} \| % {$_.Name}`
Перебор подкаталогов текущего каталога	`for /d %f in (*) do echo %f`	`dir | ? {$_.PSIsContainer} \| % {$_.Name}`
Арифметический цикл от 1 до 10	`for /l %i in (1,1,10) do echo %i`	`for ($i=1; $i -le 10; $i++) {$i}`

Не так просто обстоит дело с циклом `for /f`, который используется в оболочке cmd.exe для обработки содержимого текстовых файлов. Обработка в данном случае состоит в чтении файла, разбиении его на отдельные строки текста и выделении из каждой строки заданного числа подстрок. После этого найденные подстроки используются в качестве значений переменных при выполнении основного тела цикла (заданной команды). Рассмотрим пример. Пусть в файле c:\test.txt записаны две строки:

```
AAAAAA BBBBB CCCCCC DDDD
EE FFF GGG HHH
```

Следующий цикл в оболочке cmd.exe выделяет из каждой строки первые три слова и помещает их в переменные `%i`, `%j` и `%k`:

```
C:\> for /f "tokens=1-3" %i in (test.txt) do @echo i=%i j=%j k=%k
i=AAAAAA j=BBBBB k=CCCCCC
i=EE j=FFF k=GGG
```

Аналогичный результат в PowerShell можно получить с помощью следующего конвейера команд:

```
PS C:\> type c:\test.txt | ForEach-Object {$i, $j, $k, $l =
>> [regex]::Split($_,' '); "i=$i j=$j k=$k"}
>>
i=AAAAAA j=BBBBB k=CCCCCC
i=EE j=FFF k=GGG
```

Отметим, что в cmd.exe разделение строки на составные части работает только внутри оператора цикла `for /f`, а статический метод `[regex]::Split` с аналогичной функциональностью может применяться в PowerShell без каких-либо ограничений.

Вывод текста и запуск программ

В оболочке cmd.exe для вывода текста на экран (или перенаправления в файл) используется команда `Echo`, например:

```
C:\> echo Привет!
Привет!
```

В PowerShell для отображения текста на экране не нужно никакой специальной команды, достаточно ввести символьную строку в одиночных или двойных кавычках и нажать клавишу <Enter>, например:

```
PS C:\> "Привет!"
Привет!
```

С другой стороны, для запуска в cmd.exe исполняемого файла, содержащего пробелы в имени, достаточно указывать его имя в двойных кавычках, например:

```
C:\> "Моя программа.exe"
```

В PowerShell подобную программу можно запустить с помощью командлета `Invoke-Item` или оператора вызова `&`:

```
PS C:\> & "Моя программа.exe"
```

А исполняемые файлы, имя которых не содержит пробелов, можно вызывать так же, как и в cmd.exe.

Запуск команд cmd.exe из PowerShell

Для запуска из оболочки PowerShell внешней команды cmd.exe (например, `xcopy` или `attrib`) достаточно набрать ее имя в командной строке PowerShell и нажать клавишу <Enter>:

```
PS C:\> attrib
A          C:\09_9409284.xls
A          C:\1.bat
A          C:\1.ps1
A          C:\11.xml
A          C:\28032008.txt
A          C:\28032008.xls
A          C:\ARJ.EXE
A          C:\AUTOEXEC.BAT
```

Так же просто запускаются командные файлы с расширением `bat` или `cmd`. Пусть, например, на диске c:\ находится командный файл 1.cmd следующего содержания:

```
@echo off
echo Запущен командный файл интерпретатора cmd.exe
pause
```

Запустим его в оболочке PowerShell:

```
PS C:\> c:\1.cmd
Запущен командный файл интерпретатора cmd.exe
Для продолжения нажмите любую клавишу . . .
PS C:\>
```

Как видим, командные файлы выполняются так же, как и в оболочке cmd.exe. При этом, однако, следует учитывать, что при запуске командного файла из PowerShell создается новый процесс cmd.exe, который выполняет этот файл, после чего завершает свою работу. Поэтому все изменения, которые производились в командном файле с переменными среды, будут в конечном итоге проигнорированы.

Для запуска из PowerShell внутренней команды интерпретатора cmd.exe нужно использовать команду `cmd /c` (ключ `/c` означает режим выполнения одной команды с последующим завершением работы интерпретатора cmd.exe). Например:

```
PS C:\> cmd /c copy /?
Копирование одного или нескольких файлов в другое место.

COPY [/D] [/V] [/N] [/Y | /-Y] [/Z] [/A | /B] источник [/A | /B]
     [+ источник [/A | /B] [+ ...]] [результат [/A | /B]]

  источник    Имена одного или нескольких копируемых файлов.
  /A          Файл является текстовым файлом ASCII.
  /B          Файл является двоичным файлом.
  /D          Указывает на возможность создания зашифрованного файла

  результат   Каталог и/или имя для конечных файлов.
  /V          Проверка правильности копирования файлов.
  /N          Использование, если возможно, коротких имен при копировании
              файлов, чьи имена не удовлетворяют стандарту 8.3.
  /Y          Подавление запроса подтверждения на перезапись
              существующего конечного файла.
```

```
  /-Y        Обязательный запрос подтверждения на перезапись
             существующего конечного файла.
  /Z         Копирование сетевых файлов с возобновлением.

Ключ /Y можно установить через переменную среды COPYCMD.
Ключ /-Y командной строки переопределяет такую установку.
По умолчанию требуется подтверждение, если только команда COPY
не выполняется в пакетном файле.

Чтобы объединить файлы, укажите один конечный и несколько исходных
файлов, используя подстановочные знаки
или формат "файл1+файл2+файл3+...".
```

Как видим, в данном случае выполняется внутренняя команда `copy` интерпретатора cmd.exe, а не командлет `Copy-Item`, имеющий в PowerShell псевдоним `copy`.

Сравнение языков PowerShell и VBScript

Языки VBScript и PowerShell сильно различаются как по самой идеологии программирования, так и по синтаксису. В данном разделе мы обсудим некоторые из этих различий — это может помочь пользователям VBScript, начинающим работать с оболочкой и сценариями PowerShell.

Обращение к функциям, командам и методам

В языке VBScript параметры при вызове процедур и функций могут указываться двумя способами. Если вызывается процедура или функция, возвращающая значение, которое не присваивается никакой переменной, то подставляемые параметры указываются через пробел после имени этой процедуры или функции, а значения параметров разделяются запятыми. Например:

```
MyProcedure 3, 10
```

Если же для вызова процедуры используется оператор `Call` или возвращаемое функцией значение присваивается какой-либо переменной, то список параметров обязательно должен быть заключен в круглые скобки, например:

```
Call MyProcedure(3, 10)
a = MyFunction(3, 5)
```

В PowerShell функции всегда вызываются как команды, а не как методы, список параметров указывается через пробел после имени функции, значения параметров между собой разделяются также пробелами. Например:

```
MyFunction 3 5
```

но не

```
MyFunction(3, 5)
```

При вызове в PowerShell методов объектов их параметры разделяются между собой запятыми и указываются в круглых скобках, например:

```
$object.method(1, 2)
```

При этом между именем метода и списком параметров не должно быть пробелов (в VBScript это допускается), то есть команда

```
$object.method (1, 2)
```

вызовет ошибку. Если метод не предусматривает указания параметров, то скобки при его вызове все равно нужно ставить, например:

```
$object.method()
```

В VBScript функции могут возвращать единственное значение. Для этого нужно внутри функции присвоить нужное значение переменной, название которой совпадает с именем функции. Например:

```
Function MyFunction(Param1, Param2)
Dim Sum
  Sum = Param1+Param2
  MyFunction = Sum
End Function
```

В PowerShell функции могут неявно (без указания оператора `return` или присвоения значения переменной с именем функции) возвращать несколько значений. Любое значение, возвращаемое выражением внутри функции, добавляется в коллекцию значений, возвращаемых функцией. Инструкция `return` в PowerShell применяется только для выхода из функции до ее полного завершения. Для примера приведем функцию на языке VBScript, возвращающую символьную строку:

```
Function Hello
  Hello = "Привет!"
End Function
```

Аналогичная функция в PowerShell выглядит следующим образом:

```
Function Hello { "Привет!" }
```

Работа с переменными, массивами и объектами

В отличие от переменных в VBScript, имена переменных в PowerShell обязательно начинаются с символа `$`, например:

```
$i=0
$s="Строка"
```

В VBScript для записи в переменную ссылки на какой-либо объект нужно использовать оператор `Set`, например:

```
Set FSO=CreateObject("Scripting.FileSystemObject")
```

В PowerShell для выполнения аналогичного действия ключевое слово `Set` не указывается (напомним, что имя параметра `-ComObject` можно сократить до `-com`):

```
$FSO = New-Object -com Scripting.FileSystemObject
```

Если в VBScript обратиться к несуществующему свойству объекта, то возникнет ошибка. В PowerShell в подобной ситуации никакого сообщения об ошибке не выводится.

По умолчанию переменные в VBScript можно предварительно не объявлять; для включения режима обязательного объявления переменных нужно вставить в самую первую строку сценария выражение `Option Explicit`. В PowerShell точный аналог данного выражения отсутствует, здесь можно только запретить использование неинициализированных переменных. Данный режим включается следующей командой:

```
Set-PSDebug -Strict
```

В языке VBScript индексы для элементов массива указываются в круглых скобках после имени, например `a(10)`. В PowerShell используются квадратные скобки: `$a[10]`. Нумерация в массивах в обоих языках начинается с нуля.

Использование символьных строк

В VBScript символьные строки указываются внутри двойных кавычек, для их конкатенации (соединения) используется символ `&`, например:

```
s = "Строка 1" & "Строка 2"
```

В PowerShell строки могут заключаться как в двойные, так и в одинарные кавычки, а их конкатенация осуществляется с помощью символа `+`, например:

```
$s = 'Строка 1' + "Строка 2"
```

Если внутри строки в двойных кавычках указаны имена переменных или Escape-последовательности, то в строку подставляются их значения. Например:

```
PS C:\> $a="Привет!"
PS C:\> "Пользователь, $a"
Пользователь, Привет!
```

Прочие замечания по синтаксису

Синтаксис PowerShell похож на синтаксис С-подобных языков, так как здесь блоки кода выделяются символами { и }, а не ключевыми словами, как в VBScript. Например, в PowerShell мы должны написать

```
if ($n -eq 10) {"n=10"}
```

вместо

```
If (n=10) Then
  MsgBox "n=10"
End If
```

В PowerShell в одной строке можно размещать несколько выражений, разделяя их точкой с запятой (;). В VBScript подобное разделение выражений осуществляется с помощью двоеточия (:).

Если выражение полностью умещается на одной строке, то ни в VBScript, ни в PowerShell не нужно указывать никакого специального символа, означающего завершение строки. Если выражение не умещается на одной строке, то в VBScript его можно продолжить на следующей строке, указав в конце первой строки символ подчеркивания _. Например:

```
Text="Процесс запущен " & vbCrLf _
      & "Закрыть Блокнот?"
```

В PowerShell во многих случаях одно выражение можно размещать на нескольких строках без указания специальных знаков, например:

```
PS C:\> $a=
>> 1
>>
PS C:\> $a
1
```

Если же знак переноса на другую строку необходим, то в его качестве указывается символ обратного апострофа (`), например:

```
PS C:\> get-process -Name `
>> powershell
>>

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  ProcessName
-------  ------    -----      ----- -----   ------     --  -----------
    499       6    26420      15148   145     2.38    608  powershell
```

В VBScript и PowerShell сильно различаются между собой операторы сравнения. В VBScript используются обычные математические символы и их

комбинации (<, >, <>, <= и т. д.), а в PowerShell для каждого оператора сравнения определено свое символьное представление (-eq для оператора равенства, -ne для оператора неравенства и т. д.).

Аналоги PowerShell для функций VBScript

Предположим, что у вас есть работающие сценарии на языке VBScript, которые вы хотите переписать на язык PowerShell. Наверняка при этом вы столкнетесь с проблемой подбора в PowerShell аналогов для встроенных функций VBScript. В этом разделе рассматриваются примеры таких аналогов для некоторых наиболее часто используемых в сценариях функций, выполняющих математические операции, обработку символьной информации и манипуляции с датой и временем.

Математические функции

Практически для всех математических функций, реализованных в VBScript, можно найти аналог среди статических методов .NET-класса System.Math.

Функция *Abs*

Функция Abs(x) возвращает абсолютное значение (модуль) числа x. Для вычисления модуля числа в PowerShell можно воспользоваться статическим методом Abs класса System.Math, например:

```
PS C:\> $a=[Math]::Abs(-12)
PS C:\> $a
12
```

Функция *Atn*

Функция Atn(x) возвращает арктангенс числа x. В PowerShell для этой цели можно воспользоваться статическим методом Atan класса System.Math (результат функции возвращается в радианах), например:

```
PS C:\> $a=[Math]::Atan(1)
PS C:\> $a
0.785398163397448
```

Функции *Cos*, *Sin*, *Tan*

Функции Cos(x), Sin(x), Tan(x) возвращают косинус, синус и тангенс числа x, соответственно. В PowerShell для вычисления этих тригонометрических функций можно воспользоваться одноименными статическими методами

класса System.Math (как и в VBScript, аргумент метода указывается в радианах), например:

```
PS C:\> $a=[Math]::Cos([Math]::Pi)
PS C:\> $a
-1
```

Функции *Exp, Log*

Функция Exp(*x*) представляет собой экспоненциальную функцию, она возвращает число e, возведенное в степень *x*. Функция Log(*x*) является обратной к Exp и возвращает натуральный логарифм числа *x*. В качестве аналога этих функций в PowerShell можно использовать одноименные статические методы класса System.Math, например:

```
PS C:\> $a=[Math]::Exp(2)
PS C:\> $a
7.38905609893065
PS C:\> $b=[Math]::Log($a)
PS C:\> $b
2
```

Функция *Int*

Функция Int(*x*) возвращает целую часть числа *x*. В PowerShell для этой цели можно воспользоваться статическим методом Floor класса System.Math, например:

```
PS C:\> $a=[Math]::Floor(12.86)
PS C:\> $a
12
```

Функция *Rnd*

Функция Rnd возвращает случайное число от 0 до 1. В PowerShell данную задачу можно решить, создав экземпляр класса System.Random и воспользовавшись методом NextDouble данного класса:

```
PS C:\> $a=New-Object Random
PS C:\> $a.NextDouble()
0.189007192006804
```

Функция *Round*

Функция Round(*x* [, *numdecimal*]) возвращает результат округления числа *x* с точностью до *numdecimal* знаков после запятой. В PowerShell в качестве

аналога данной функции можно использовать одноименный статический метод класса `System.Math`, например:

```
PS C:\> $a=[Math]::Round(10.78,1)
PS C:\> $a
10.8
```

Функция *Sgn*

Функция `Sgn(x)` — знаковая функция числа *x* (которая равна –1, если число *x* меньше нуля, равна 0, если *x* равно 0, и равна 1, если *x* больше нуля). В PowerShell в качестве знаковой функции можно использовать статический метод `Sign` класса `System.Math`, например:

```
PS C:\> $a=[Math]::Sign(-560)
PS C:\> $a
-1
```

Функция *Sqr*

Функция `Sqr(x)` вычисляет квадратный корень из числа *x* и возвращает полученное значение. В PowerShell вычислить квадратный корень можно с помощью статического метода `Sqrt` класса `System.Math`, например:

```
PS C:\> $a=[Math]::Sqrt(169)
PS C:\> $a
13
```

Символьные функции

Функции, обрабатывающие символьную информацию, используются в сценариях VBScript очень часто. Для символьных функций VBScript не так просто указать аналог из мира PowerShell, как для математических: здесь могут потребоваться статические и обычные методы класса `System.String`, а также некоторые встроенные операторы PowerShell.

Функция *Asc*

Функция `Asc(str)` возвращает ASCII-код первого символа в строке *str*. Для определения ASCII-кода символа в PowerShell можно дважды выполнить преобразование типов, например:

```
PS C:\> $a=[byte][char] "A"
```

В этом примере в переменную `$a` заносится код буквы "A" (число 65):

```
PS C:\> $a
65
```

Функция *Chr*

Функция `Chr(code)` является обратной к `Asc`: она возвращает символ с ASCII-кодом *code*. Для выполнения этой операции достаточно преобразовать числовое значение кода символа к типу `char`:

```
PS C:\> $a=[char]65
```

После выполнения данной команды переменная `$a` будет содержать символ "A":

```
PS C:\> $a
A
```

Функция *InStr*

Функция `InStr` используется для поиска вхождения подстроки в строку. Чаще всего данная функция вызывается с двумя символьными параметрами, первый из которых задает строку, в которой производится поиск, а второй — искомую подстроку. При этом функция возвращает индекс символа, с которого начинается первое вхождение подстроки в строку (нумерация начинается с единицы). Если вхождение не найдено, то возвращается 0. Например, функция `InStr("abcdef", "cd")` вернет число 3.

В PowerShell подобную функциональность обеспечивает метод `IndexOf`, имеющийся у строк. Если вхождение найдено, то данный метод возвращает индекс символа, с которого начинается первое вхождение подстроки в строку (в отличие от VBScript, нумерация при этом начинается с нуля). Если вхождение не найдено, то возвращается число –1. Например:

```
PS C:\> $a="abcdef"
PS C:\> $b=$a.IndexOf("cd")
PS C:\> $b
2
```

Функция *Join*

Функция `Join(list[, delim])` возвращает строку, полученную в результате конкатенации подстрок, содержащихся в массиве *list*. Параметр *delim* задает символ, разделяющий подстроки (по умолчанию таким символом является пробел).

В PowerShell можно получить аналогичную функциональность, воспользовавшись статическим методом `Join` класса `System.String`. Например, зададим массив `$a`:

```
PS C:\> $a="П", "р", "и", "в", "е", "т"
```

Выполним следующую команду:

```
PS C:\> $b=[string]::Join(" ", $a)
```

Переменная $b будет содержать строку, полученную в результате конкатенации элементов массива $a с пробелом в качестве разделителя:

```
PS C:\> $b
П р и в е т
```

Функция *LCase*

Функция LCase(*str*) возвращает строку, в которой все алфавитные символы преобразованы к нижнему регистру. В PowerShell для этого достаточно вызвать метод ToLower, имеющийся у строк. Например:

```
PS C:\> $a="ПРИВЕТ"
PS C:\> $b=$a.ToLower()
PS C:\> $b
привет
```

Функция *Left*

Функция Left(*str, len*) возвращает *len* символов с начала строки *str*. Например, функция вернет Left("Привет", 2) строку "Пр".

В PowerShell выделить подстроку из строки можно с помощью метода Substring, имеющегося у строк. В качестве первого параметра этого метода указывается номер начальной позиции подстроки (для нашего случая это ноль), а в качестве второго — количество выделяемых символов. Например, после выполнения команд

```
PS C:\> $a="Привет"
PS C:\> $b=$a.Substring(0,2)
```

в переменной $b будет храниться строка "Пр":

```
PS C:\> $b
Пр
```

Функция *Len*

Функция Len(*str*) возвращает число символов в строке *str*. В PowerShell количество символов в строке хранится в ее свойстве Length, например:

```
PS C:\> $a="Привет"
PS C:\> $b=$a.Length
PS C:\> $b
6
```

Функции *LTrim*, *RTrim* и *Trim*

Функции `LTrim(str)`, `RTrim(str)`, `Trim(str)` удаляют из строки *str* начальные, конечные или и те и другие пробелы, соответственно. Аналоги этих функций в PowerShell — методы `TrimStart` (удаление пробелов слева), `TrimEnd` (удаление пробелов справа) и метод `Trim` (удаление всех пробелов справа и слева). Например, поместим в переменную `$a` строку, содержащую начальные и конечные пробелы:

```
PS C:\> $a="   Привет   "
```

Применим теперь указанные выше методы, помещая результат в переменную `$b`. Для проверки будем выводить значение переменной `$b`, ограниченное символами "*":

```
PS C:\> $b=$a.TrimStart()
PS C:\> "*"+$b+"*"
*Привет   *
PS C:\> $b=$a.TrimEnd()
PS C:\> "*"+$b+"*"
*   Привет*
PS C:\> $b=$a.Trim()
PS C:\> "*"+$b+"*"
*Привет*
```

Функция *Mid*

Функция `Mid(str, start, len)` возвращает из строки *str* подстроку, которая начинается с позиции *start* и имеет длину *len*. В PowerShell для этой цели следует пользоваться методом `Substring`, имеющимся у строк. Например, зададим значение символьной переменной `$a`:

```
PS C:\> $a="Привет"
```

Для получения, скажем, трех символов, начиная со второго, нужно выполнить следующую команду:

```
PS C:\> $b=$a.Substring(1,3)
```

Первый параметр здесь задает номер начального символа (нумерация символов в строках PowerShell идет с нуля), второй параметр — количество извлекаемых символов. Проверим значение переменной `$b`:

```
PS C:\> $b
рив
```

Функция *Replace*

Функция `Replace(expr, find, replacewith)` возвращает строку, которая получается из строки *expr* путем замен входящих в нее подстрок *find* на подстроки *replacewith*. В PowerShell замену подстрок позволяет выполнить оператор `-replace`. Например, сохраним в переменной `$a` символьную строку

```
PS C:\> $a="Привет, Андрей!"
```

и поменяем в `$a` подстроку "Привет" на "Пока":

```
PS C:\> $b=$a -replace "Привет", "Пока"
PS C:\> $b
Пока, Андрей!
```

Функция *Right*

Функция `Right(str, len)` возвращает *len* символов с конца строки *str*. Получить тот же результат в PowerShell можно с помощью метода `Substring`, в качестве параметров которого нужно указать индекс первого символа и количество символов в возвращаемой подстроке. Для вычисления индекса начального символа нужно из длины строки (значение свойства `Length`) вычесть количество копируемых символов. Рассмотрим пример. Запишем в переменную `$a` строку:

```
PS C:\> $a="Привет"
```

Выделим из этой строки 2 последних символа:

```
PS C:\> $b=$a.Substring($a.Length-2,2)
PS C:\> $b
ет
```

Функция *Space*

Функция `Space(n)` возвращает строку, состоящую из *n* пробелов. Для получения подобной строки в PowerShell можно просто воспользоваться оператором умножения. Например, команда

```
PS C:\> $a=" "*10
```

запишет в переменную `$a` строку из десяти пробелов.

Функция *Split*

Функция `Split(Expr[, delim])` возвращает массив строк, полученных в результате разбиения строки *Expr* на подстроки. Параметр *delim* задает символ, разделяющий подстроки (по умолчанию таким символом является пробел).

Строки в PowerShell имеют метод `Split` с той же функциональностью. Рассмотрим пример. Сохраним в переменной `$a` строку "Раз,два,три,четыре":

```
PS C:\> $a="Раз,два,три,четыре"
```

Выделим из переменной `$a` подстроки, разделенные запятой, поместив результат в переменную `$b`:

```
PS C:\> $b=$a.Split(",")
```

Теперь переменная `$b` является массивом, содержащим четыре элемента:

```
PS C:\> $b
Раз
два
три
четыре
```

Функция *StrComp*

Функция `StrComp(str1, str2)` возвращает число — результат сравнения строк *str1* и *str2*. Если *str1*<*str2*, то возвращается −1; если *str1*=*str2*, то возвращается 0; если *str1*>*str2*, то возвращается 1. В PowerShell с помощью статического метода `Compare` класса `System.String` можно определить, равны ли две строки. Данный метод имеет три параметра. Первые два параметра задают сравниваемые строки, а третий параметр логического типа определяет режим сравнения строк: с учетом регистра символов (`$False`) или без учета (`$True`). Например:

```
PS C:\> $a="привет"
PS C:\> $b="ПРИВЕТ"
PS C:\> $c=[String]::Compare($a, $b, $True)
```

В этом случае сравнение строк `$a` и `$b` производится без учета регистра символов, и метод `Compare` возвращает ноль, что означает, что сравниваемые строки равны:

```
PS C:\> $c
0
```

Если же сравнить эти строки с учетом регистра, то метод `Compare` вернет число −1, что означает, что сравниваемые строки не равны:

```
PS C:\> $c=[String]::Compare($a, $b, $false)
PS C:\> $c
-1
```

Функция *String*

Функция `String(number, char)` возвращает строку, состоящую из *number* символов *char*. В PowerShell подобные строки можно формировать с помощью оператора умножения, например:

```
PS C:\> $a="+" * 10
```

После выполнения данной команды в переменной `$a` будет записана строка из десяти символов "+":

```
PS C:\> $a
++++++++++
```

Функция *UCase*

Функция `UCase(str)` возвращает строку, в которой все алфавитные символы преобразованы к верхнему регистру. В PowerShell для этого достаточно вызвать метод `ToUpper`, имеющийся у строк. Например:

```
PS C:\> $a="привет"
PS C:\> $b=$a.ToUpper()
PS C:\> $b
ПРИВЕТ
```

Функции для работы с датами и временем

Последний набор встроенных функций языка VBScript, которые мы рассмотрим в этом разделе, — это функции, связанные с манипуляциями с датами и временем.

Функция *Date*

Функция `Date` возвращает текущую системную дату (без системного времени). В PowerShell для этой цели можно выполнить командлет `Get-Date` с параметром `-Format`, равным "d":

```
PS C:\> Get-Date -Format d
19.05.2008
```

Функция *DateAdd*

Функция `DateAdd(interval, number, date)` возвращает дату, отстоящую от даты *date* на *number* интервалов, заданных параметром *interval*, который может принимать следующие значения: "yyyy" — год, "q" — квартал, "m" — месяц, "w" — неделя, "ww" — неделя года, "y" — день года, "d" — день, "h" — час, "m" — минута, "s" — секунда. В PowerShell подобную задачу

можно решить с помощью соответствующего метода класса System.DateTime, экземпляр которого создается при выполнении командлета Get-Date (напомним, что gm — это псевдоним командлета Get-Member):

```
PS C:\> Get-Date | gm

   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(TimeSpan ...
AddDays              Method         System.DateTime AddDays(Doubl...
AddHours             Method         System.DateTime AddHours(Doub...
AddMilliseconds      Method         System.DateTime AddMillisecon...
AddMinutes           Method         System.DateTime AddMinutes(Do...
AddMonths            Method         System.DateTime AddMonths(Int...
AddSeconds           Method         System.DateTime AddSeconds(Do...
AddTicks             Method         System.DateTime AddTicks(Int6...
AddYears             Method         System.DateTime AddYears(Int3...
. . .
```

Например, для сохранения в переменной $a даты, отстоящей от текущей на 38 дней, нужно выполнить следующую команду:

```
PS C:\> $a=(Get-Date).AddDays(38)
```

Проверим значение переменной $a:

```
PS C:\> $a

26 июня 2008 г. 22:00:39
```

Функция *DateDiff*

Функция DateDiff(*interval*, *date1*, *date2*) возвращает разницу в интервалах *interval* (возможные значения этого параметра те же, что и в функции DateAdd) между датами *date1* и *date2*. В PowerShell аналогичной функциональностью обладает командлет New-TimeSpan.

Например, следующая команда заносит в переменную $a разницу между текущей датой (значение параметра -End, в нашем примере это 19 мая 2008 года) и первым января 2008 года (значение параметра -Start):

```
PS C:\> $a=New-TimeSpan $(Get-Date -Month 1 -Day 1 -Year 2008)
```

Проверим значение переменной `$a`:

```
PS C:\> $a

Days              : 139
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 0
Ticks             : 120096000000000
TotalDays         : 139
TotalHours        : 3336
TotalMinutes      : 200160
TotalSeconds      : 12009600
TotalMilliseconds : 12009600000
```

Таким образом, если нас интересует количество дней, разделяющее две даты, можно обратиться к значению свойства `Days`:

```
PS C:\> $a.days
139
```

Функция *DatePart*

Функция `DatePart(interval, date)` возвращает ту часть даты *date*, которая соответствует параметру *interval* (параметр принимает те же значения, что и в функции `DateAdd`). В PowerShell для той же цели служат свойства объекта `System.DateTime`, возвращаемого командлетом `Get-Date`, например:

```
PS C:\> $d=Get-Date
PS C:\> $d
19 мая 2008 г. 14:37:15
PS C:\> $d.Day
19
PS C:\> $d.DayOfWeek
Monday
PS C:\> $d.DayOfYear
140
PS C:\> $d.Hour
14
PS C:\> $d.MilliSecond
437
```

```
PS C:\> $d.Minute
37
PS C:\> $d.Month
5
PS C:\> $d.Second
15
PS C:\> $d.Year
2008
```

Функция *DateSerial*

Функция `DateSerial(year, month, day)` возвращает переменную типа `Date`, которая соответствует указанным году (параметр *year*), месяцу (параметр *month*) и дню (параметр *day*). К аналогичному результату в PowerShell приводит вызов командлета `Get-Date` с параметрами `-Year`, `-Month` и `-Day`, задающими год, месяц и день соответственно, например:

```
PS C:\> $a=Get-Date -Year 2008 -Month 10 -Day 30
PS C:\> $a

30 октября 2008 г. 22:54:34
```

Функция *DateValue*

Функция `DateValue(date)` возвращает переменную типа `Date`, которая соответствует дате, заданной символьным параметром *date*. В PowerShell можно явно преобразовать символьное значение к типу `DateTime`, например:

```
PS C:\> $a=[DateTime] "10/30/2008"
```

Проверим тип и значение переменной `$a`:

```
PS C:\> $a.GetType()

IsPublic IsSerial Name
-------- -------- ----
True     True     DateTime
PS C:\> $a

30 октября 2008 г. 0:00:00
```

Функции *Day*, *Month*, *Year*, *Hour*, *Minute*

Функции `Day(date)`, `Month(date)`, `Year(date)`, `Hour(date)`, `Minute(date)` возвращают целое число, соответствующее порядковому номеру, соответст-

венно, дня, месяца, года, часа и минуты в дате *date*. В PowerShell для определения этих компонент даты можно воспользоваться значением одноименных свойств объекта System.DateTime, возвращаемого командлетом Get-Date. Эти свойства нам уже встречались, когда мы говорили о замене функции DatePart. Например:

```
PS C:\> $a=(Get-Date).Year
PS C:\> $a
2008
```

Функция *IsDate*

Функция IsDate(*expr*) возвращает True, если параметр *expr* задает корректную дату, и False в противном случае. В PowerShell для проверки соответствия переменной определенному типу следует использовать оператор -is, например:

```
PS C:\> $a=Get-Date
PS C:\> $a -is [datetime]
True
```

Функция *MonthName*

Функция MonthName(*month*) возвращает наименование для месяца с номером *month*. Для определения наименования месяца в PowerShell можно воспользоваться параметром -Format (или его сокращением -f) командлета Get-Date, указав в качестве этого параметра аббревиатуру "MMMM", например:

```
PS C:\> $a=Get-Date -f "MMMM"
PS C:\> $a
Май
```

Функция *Now*

Функция Now возвращает текущие дату и время в виде, соответствующем региональным настройкам Windows. В PowerShell для этой цели используется командлет Get-Date:

```
PS C:\> $a=Get-Date
PS C:\> $a

19 мая 2008 г. 23:44:23
```

Функция *Time*

Функция `Time` возвращает текущее системное время. В PowerShell для этого можно воспользоваться командлетом `Get-Date` с параметром `-DisplayHint`, имеющим значение `time`:

```
PS C:\> $a=Get-Date -DisplayHint time
PS C:\> $a

23:50:01
```

> ***Замечание***
>
> Параметр `-DisplayHint` влияет на режим отображения, но возвращаемый объект по-прежнему содержит полную информацию о дате.

Функция *Timer*

Функция `Timer` возвращает количество секунд, прошедших с полуночи. Обычно эта функция используется для оценки длительности выполнения сценария, функции или какого-либо другого фрагмента кода. В PowerShell подобной функциональностью обладает командлет `Measure-Command`, измеряющий время выполнения команд. Например, оценим время выполнения цикла `for`:

```
PS C:\> $a=Measure-Command {for ($a=1; $a -le 1000000; $a++) {$a}}
PS C:\> $a

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 15
Milliseconds      : 209
Ticks             : 152094995
TotalDays         : 0.000176035873842593
TotalHours        : 0.00422486097222222
TotalMinutes      : 0.253491658333333
TotalSeconds      : 15.2094995
TotalMilliseconds : 15209.4995
```

Если же для каких-то целей нужно узнать именно количество секунд, прошедших с полуночи, можно воспользоваться следующей командой:

```
PS C:\> (Get-Date).TimeOfDay.TotalSeconds
52635,4375
```

Функция *TimeValue*

Функция `TimeValue(time)` возвращает переменную типа `Date`, которая соответствует времени, заданному символьным параметром *time*. В PowerShell можно явно преобразовать символьное значение к типу `DateTime`, например:

```
PS C:\> $a=[datetime] "12:30:00"
```

Проверим тип и значение переменной `$a` (в данном примере предполагается, что предыдущая команда была выполнена 12 мая 2008 г.):

```
PS C:\> $a.GetType()

IsPublic IsSerial Name
-------- -------- ----
True     True     DateTime
PS C:\> $a
20 мая 2008 г. 12:30:00
```

Функция *Weekday*

Функция `Weekday(date)` возвращает целое число — день недели для даты, заданной параметром *date* (по умолчанию воскресенью соответствует число 0, понедельнику — число 1, вторнику — число 2 и т. д.). В PowerShell класс `System.DateTime`, экземпляр которого возвращается командлетом `Get-Date`, содержит свойство `DayOfWeek`, однако в этом свойстве хранится символьное обозначение дня недели. Для получения числового номера можно выполнить следующую команду (после `value` указывается два символа подчеркивания):

```
PS C:\> $a=(Get-Date).DayOfWeek.value__
PS C:\> $a
2
```

Функция *WeekdayName*

Функция `WeekdayName(weekday)` возвращает наименование для дня недели с порядковым номером *weekday*. В PowerShell для этого достаточно обратиться к свойству `DayOfWeek` объекта `System.DateTime`, возвращаемого командлетом `Get-Date`, например:

```
PS C:\> (Get-Date).DayOfWeek
Tuesday
```

Использование из PowerShell кода VBScript

В предыдущих разделах обсуждались вопросы перевода имеющихся сценариев VBScript на язык PowerShell. Естественно, совсем не обязательно переводить все работающие у вас сценарии VBScript на новый язык, вы можете спокойно запускать готовые сценарии, работая в оболочке PowerShell, и даже интегрировать (встраивать) фрагменты кода на языке VBScript в сценарии PowerShell.

Для выполнения готового сценария VBScript в оболочке PowerShell, как и в cmd.exe, достаточно указать его имя в командной строке и нажать клавишу <Enter>, например:

```
PS C:\> eventquery.vbs /?
Сервер сценариев Windows (Microsoft R) версия 5.6
с Корпорация Майкрософт (Microsoft Corp.), 1996-2001. Все права защищены.

EVENTQUERY.vbs [/S <система> [/U <пользователь> [/P <пароль>]]] [/FI
<фильтр>] [/FO <формат>] [/R <диапазон>] [/NH] [/V] [/L <журнал>  | *] .
. . .
```

Кроме этого, с помощью COM-объекта `ScriptControl` можно встроить фрагмент кода на языке VBScript непосредственно в сценарий PowerShell. Рассмотрим пример подобного встраивания VBScript-кода. Сначала создадим экземпляр COM-объекта `ScriptControl` и поместим ссылку на этот объект в переменную `$sc`:

```
PS C:\> $sc=New-Object -com ScriptControl
```

В свойство `Language` данного объекта следует занести идентификатор, определяющий используемый язык сценариев (в нашем случае это строка 'VBScript'):

```
PS C:\> $sc.Language='VBScript'
```

С помощью метода `AddCode` опишем функцию `GetLength` на языке VBScript, которая в дальнейшем будет доступна из PowerShell (при необходимости можно определить несколько функций):

```
PS C:\> $sc.AddCode('
>> Function GetLength(s)
>>   GetLength=Len(s)
>> End Function
>> ')
>>
```

Присвоим теперь переменной $vb свойство CodeObject объекта $sc:

```
PS C:\> $vb=$sc.CodeObject
```

Переменная $vb является объектом, содержащим в качестве метода нашу функцию GetLength:

```
PS C:\> $vb | Get-Member

   TypeName: System.__ComObject#{c59c6b12-f6c1-11cf-8835-00a0c911e8b2}

Name      MemberType Definition
----      ---------- ----------
GetLength Method     Variant GetLength (Variant)
```

Вызовем функцию (метод) GetLength, передав ей в качестве параметра строку "Привет!":

```
PS C:\> $vb.GetLength("Привет!")
7
```

Как видим, VBScript-функция GetLength корректно определила длину переданной ей строки.

Для удобства выполненную нами процедуру преобразования функции на языке VBScript в метод объекта можно оформить в виде функции PowerShell Call-VBScript, которая будет возвращать ссылку на нужный объект (листинг 18.1).

Листинг 18.1. Функция Call-VBScript

```
function Call-VBScript {
  $sc = New-Object -ComObject ScriptControl
  $sc.Language = 'VBScript'
  $sc.AddCode('
    Function GetLength(s)
      GetLength = Len(s)
    End Function
  ')
  $sc.CodeObject
  }
```

После такого определения функции в PowerShell можно инициализировать объект, который будет содержать метод GetLength, просто вызывая функцию Call-VBScript:

```
PS C:\> $vb=Call-VBScript
PS C:\> $vb.GetLength("Привет!")
7
```

Использование из PowerShell кода JScript

Сервер сценариев Windows Script Host позволяет выполнять сценарии, написанные не только на языке VBScript, но и на любом другом языке, поддерживающем технологию ActiveX Scripting. Вторым после VBScript стандартным языком для написания сценариев Windows является JScript (реализация спецификации ECMA Script (JavaScript) от компании Microsoft). С помощью COM-объекта `ScriptControl` в сценарий PowerShell можно встроить фрагмент кода и на языке JScript. Делается это аналогично рассмотренному ранее случаю с кодом VBScript, только в свойство `Language` объекта `ScriptControl` следует поместить строку 'JScript'.

В листинге 18.2 представлена функция `Call-JScript` для формирования объекта, содержащего в качестве метода JScript-функцию `getLength`.

Листинг 18.2. Функция Call-JScript

```
function Call-JScript {
  $sc = New-Object -ComObject ScriptControl
  $sc.Language = 'JScript'
  $sc.AddCode('
    function getLength(s) {
      return s.length
      }
  ')
  $sc.CodeObject
  }
```

Для вызова функции `getLength` нужно предварительно создать и инициализировать переменную `$js`, присвоив ей значение функции `Call-JScript`:

```
PS C:\> $js = Call-JScript
```

Теперь можно вызвать метод `getLength`:

```
PS C:\> $js.getLength("Привет!")
7
```

Заключение

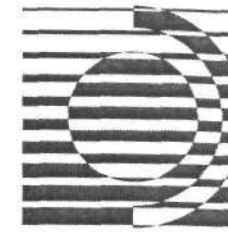
До последнего времени было широко распространено мнение о том, что сценарии (скрипты), выполняемые в операционной системе Microsoft Windows, значительно проигрывают в универсальности, гибкости и мощи аналогичным сценариям в UNIX-системах. Однако в последние годы компания Microsoft разработала несколько технологий (WSH, WMI, ADSI) и программных продуктов, позволяющих ликвидировать отставание в данной области. Последний шаг в этом направлении — новая оболочка Windows PowerShell, поддерживающая собственный мощный и в то же время простой в изучении объектно-ориентированный язык программирования и дающая возможность, среди прочего, работать из командной строки с инфраструктурами COM, WMI и .NET Framework.

Изучение PowerShell, несомненно, будет полезно разным категориям пользователей и специалистов, работающим в операционной системе Windows.

- Опытные пользователи с помощью PowerShell смогут повысить эффективность своей повседневной работы в Windows, автоматизируя часто выполняемые операции с файлами, документами, сетевыми ресурсами.
- Администраторам операционной системы PowerShell пригодится в качестве удобного и мощного инструмента для управления Windows-системами (обращение из оболочки или сценариев PowerShell к объектам WMI позволяет удаленно администрировать рабочие станции и серверы, причем делать это намного проще, чем с помощью сценариев WSH).
- Специалисты-разработчики программ смогут, используя PowerShell, ускорить реализацию интерфейсов управления к своим приложениям.

Разумеется, для грамотной работы с оболочкой и сценариями PowerShell необходимо понимать основные принципы обработки объектов в системе, знать возможности языка PowerShell, уметь использовать внешние объектные модели. В книге мы попытались пояснить основные идеи, механизмы и конструкции языка PowerShell, а также дать примеры использования команд и сценариев PowerShell в практической работе администраторов Windows. Надеемся, что, изучив эти основы, вы научитесь быстро и элегантно решать любые возникающие перед вами задачи по автоматизации работы с помощью средств PowerShell.

ПРИЛОЖЕНИЯ



Приложение 1

Объектная модель WMI

Напомним, что одной из первых и основных задач при разработке новой оболочки командной строки Windows было обеспечение удобного доступа к инфраструктуре WMI — глобального инструментария для настройки, управления и слежения за работой различных частей корпоративной компьютерной сети. В частности, используя WMI, можно с помощью специальных утилит или сценариев (на языках VBScript или PowerShell) решать целый ряд задач.

- **Управление различными версиями операционной системы Windows.** С помощью сценариев WMI можно обращаться к системным счетчикам производительности, анализировать журналы событий (Event Logs), работать с файловой системой и установленными принтерами, управлять запущенными процессами и сервисами, просматривать и изменять настройки реестра, создавать и удалять совместно используемые ресурсы и т. д. При этом все операции можно выполнять единообразно как на локальной, так и на удаленной машине.
- **Управление ресурсами и службами сети.** Сценарии WMI позволяют настраивать сетевые службы (DNS, DHCP и т. п.) и управлять сетевыми устройствами, поддерживающими технологию SNMP (Simple Network Management Protocol).
- **Мониторинг состояния системы в реальном времени.** Можно создавать сценарии-обработчики событий WMI, которые позволяют отслеживать и нужным образом обрабатывать те или иные изменения в информационной системе (например, появление определенной записи в журнале событий на локальном или удаленном компьютере, заполнение жесткого диска сервера до заданного предела, изменение некоторого ключа в системном реестре и т. п.).
- **Управление серверными приложениями Windows.** С помощью WMI можно управлять различными приложениями Microsoft: Application Center,

Operations Manager, Systems Management Server, Internet Information Server, Exchange Server, SQL Server.

Учитывая особое место технологии WMI среди инструментов и средств автоматизации Windows, мы рассмотрим более подробно структуру данной объектной модели и утилиты для интерактивной работы с ней.

Общая структура WMI

Инструментарий WMI состоит из трех частей, показанных на рис. П1.1:

1. **Управляемые объекты/ресурсы** (managed resources) — любые логические или физические компоненты информационной системы, доступ к которым может быть получен с помощью WMI. В качестве управляемых ресурсов могут выступать, например, файлы на жестком диске, запущенный экземпляр приложения, системное событие, предоставленный в общее пользование ресурс, сетевой пакет или установленный в компьютере процессор.
2. **Ядро WMI** (WMI infrastructure). Это связующее звено архитектуры WMI, отвечающее за связь управляющих программ с управляемыми объектами. Ядро WMI, в свою очередь, можно разделить на три части: *менеджер объектов CIM* (Common Information Model Object Manager, CIMOM), *репозиторий* (хранилище классов и объектов) *CIM* и *провайдеры WMI*. Кроме этого, для доступа к WMI с помощью сценариев необходима специальная *библиотека поддержки сценариев WMI* (WMI scripting library), которая располагается в файле wbemdisp.dll в каталоге %SystemRoot%\System32\Wbem.
3. **Управляющие программы** (management applications), которые являются *потребителями* сервисов WMI. В качестве потребителей могут выступать полновесные Win32-приложения, Web-приложения, сценарии WSH или другие инструменты администрирования, с помощью которых происходит доступ к управляемым объектам посредством WMI.

Отметим, что управляющие программы (потребители) различных типов применяют разные механизмы для доступа к WMI, то есть используют разные интерфейсы прикладного программирования (Application Programming Interface, API). Программы Win32 могут взаимодействовать с WMI напрямую, используя для этого WMI COM API — главный API управления. ActiveX-компоненты WMI реализуют API другого уровня: разработчики Web-приложений применяют средства управления ActiveX для создания сетевых интерфейсов к данным WMI. Еще один способ управления WMI предполагает использование сценариев WSH с помощью специального API WMI для сценариев (такие сценарии мы иногда будем называть просто сценариями WMI).

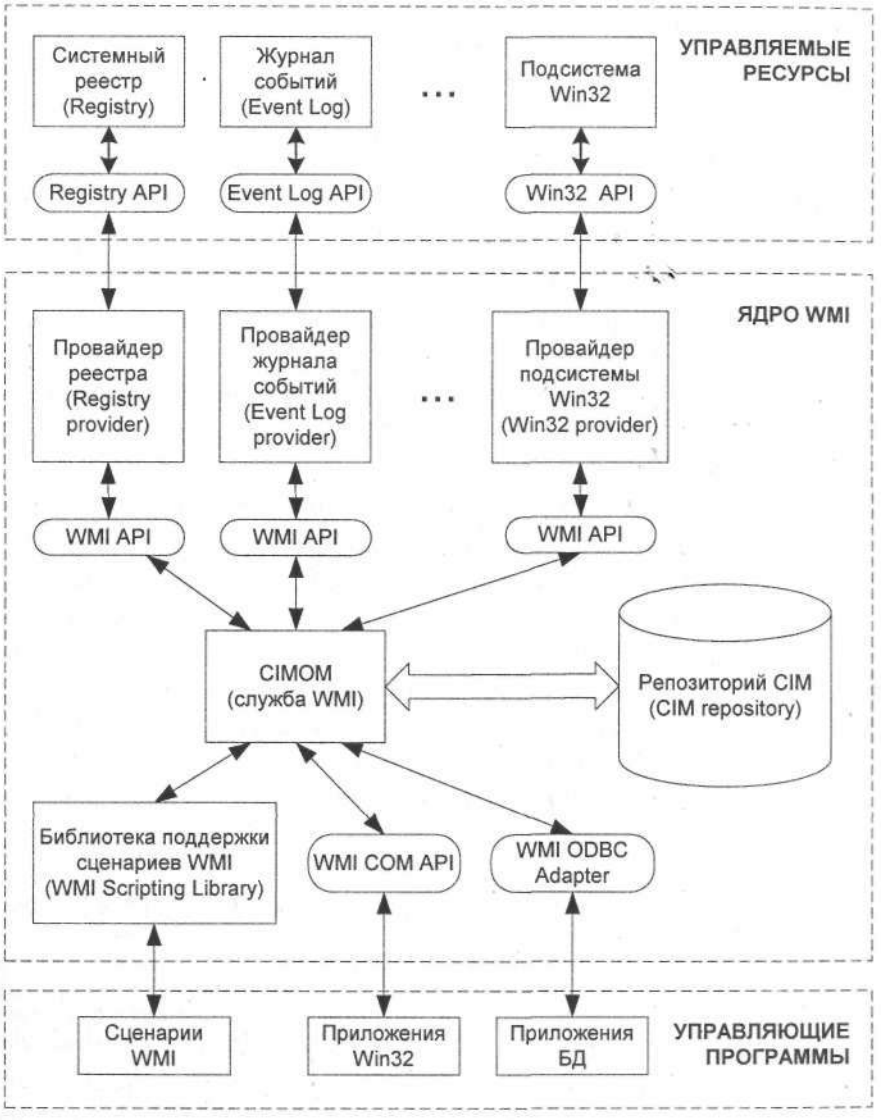


Рис. П1.1. Архитектура WMI

Ядро WMI

Ядро WMI составляют провайдеры WMI, менеджер объектов CIM и репозиторий CIM. Рассмотрим эти компоненты более подробно.

Провайдеры WMI

Провайдеры WMI обеспечивают связь между менеджером объектов CIM и управляемыми ресурсами: провайдеры предоставляют менеджеру объектов данные об управляемом объекте, обрабатывают запросы от управляющих программ и генерируют сообщения о наступлении определенных событий (см. рис. П1.1).

При этом провайдер WMI общается с управляемым объектом с помощью специфического API этого объекта, а с CIMOM — посредством стандартного интерфейса прикладного программирования WMI (WMI API). Таким образом, провайдеры скрывают детали внутренней реализации управляемых объектов, позволяя CIMOM обращаться к этим объектам единообразно, используя один и тот же WMI API.

Фактически провайдеры WMI являются серверами COM или DCOM, которые представлены динамическими библиотеками (DLL), находящимися чаще всего в каталоге %SystemRoot%\System32\Wbem. WMI включает в себя множество встроенных (стандартных) провайдеров для операционных систем Windows 2000, Windows XP, Windows Server 2003 и Windows Vista, которые предназначены для получения данных из известных системных источников таких, как подсистема Win32, журналы событий, системный реестр, системные счетчики производительности. В табл. П1.1 приведено описание некоторых стандартных провайдеров, которые присутствуют во всех перечисленных операционных системах.

Таблица П1.1. *Некоторые стандартные провайдеры WMI*

Провайдер	DLL-файл	Описание
Провайдер каталога Active Directory (Active Directory provider)	Dsprov.dll	Позволяет обращаться к объектам Active Directory как к объектам WMI
Провайдер журнала событий (Event Log provider)	Ntevt.dll	Обеспечивает управление журналом событий (предоставляя возможность выборки по определенному критерию записей для чтения, создания резервных копий и очистки журнала, изменения настроек и т. д.). Также этот провайдер позволяет обрабатывать события, генерируемые журналом (например, добавление в журнал записи определенного типа)

Таблица П1.1 *(окончание)*

Провайдер	DLL-файл	Описание
Провайдер системных счетчиков производительности (Performance Counter provider)	Wbemperf.dll	Обеспечивает доступ к счетчикам производительности, то есть к данным, позволяющим численно оценивать производительность системы
Провайдер реестра (Registry provider)	Stdprov.dll	Позволяет читать данные из реестра, создавать и модифицировать в нем ключи и разделы. Кроме этого, провайдер может генерировать события WMI при изменении определенного ключа или ветви реестра
Провайдер SNMP-устройств (SNMP provider)	Snmpincl.dll	Является шлюзом для доступа к системам и устройствам, которые управляются с помощью протокола SNMP (Simple Network Management Protocol)
Провайдер драйверов устройств (WDM provider)	Wmiprov.dll	Позволяет получить доступ к информации низкого уровня о драйверах устройств Windows Driver Model (WDM); в качестве таких устройств могут выступать, например, порты ввода/вывода или сетевые платы
Провайдер подсистемы Win32 (Win32 provider)	Cimwin32.dll	Обеспечивает доступ к информации о компьютере, операционной системе, подсистеме безопасности, дисках, периферийных устройствах, файловых системах, файлах, папках, сетевых ресурсах, принтерах, процессах, сервисах и т. п.
Провайдер инсталлированных программных продуктов (Windows Installer provider)	Msiprov.dll	Позволяет получить информацию об инсталлированном программном обеспечении

Технология WMI позволяет также создавать и устанавливать провайдеры сторонних поставщиков, с помощью которых можно будет, скажем, через WMI получать информацию о каких-то специфических устройствах или службах. Отметим, что собственные провайдеры WMI имеются в таких продуктах Microsoft, как Application Center, Operations Manager, Systems Management Server, Internet Information Server, Exchange Server, SQL Server.

Менеджер объектов CIM

Задачей менеджера объектов CIM (CIMOM) является обеспечение взаимодействия между потребителями сервисов WMI (управляющими приложениями)

и провайдерами WMI (см. рис. П1.1). CIMOM обрабатывает все запросы, которые поступают от управляющих приложений к WMI, и обеспечивает доставку к этим приложениям информации, полученной в результате выполнения таких запросов. Рассмотрим подробнее функции, выполняемые CIMOM.

- **Регистрация провайдеров.** Все провайдеры WMI должны быть зарегистрированы с помощью CIMOM; информация о провайдере (например, тип этого провайдера или путь к библиотеке DLL, которой он представлен) хранится в репозитории CIM.
- **Переадресация запросов.** Используя информацию о зарегистрированных провайдерах, CIMOM перенаправляет полученный от управляющего приложения запрос к нужному провайдеру.
- **Доступ к удаленной машине с WMI.** Управляющее приложение может обратиться с запросом к любой удаленной машине, на которой установлен WMI. При этом происходит соединение с CIMOM на удаленной машине, после чего все запросы обрабатываются точно так же, как и на локальной машине.
- **Обеспечение безопасности.** Защита ресурсов WMI состоит в том, что CIMOM проверяет права пользователя, который пытается воспользоваться сервисами WMI на локальном или удаленном компьютере.
- **Обработка запросов управляющих приложений.** Потребители WMI обращаются к управляемым объектам с помощью специального языка запросов WMI Query Language (WQL). Если провайдер запрашиваемого объекта не поддерживает напрямую WQL, то CIMOM должен преобразовать этот запрос к тому виду, в котором он сможет быть обработан этим провайдером.
- **Обработка событий WMI.** Поддержка CIMOM этой функции позволяет потребителям WMI создавать обработчики событий, которые возникают при определенном изменении в управляемом объекте (примеры таких событий — снижение объема свободного пространства на жестком диске до заданного значения или запуск на компьютере определенного приложения). Для этого CIMOM периодически опрашивает нужный объект (интервал опроса задается управляющим приложением) и генерирует событие, как только обнаруживает, что заданное заранее условие выполнено.

В Windows функциональность менеджера CIM обеспечивает файл winmgmt.exe, который находится в каталоге %SystemRoot%\System32\Wbem (этот файл запускается как сервис).

Репозиторий CIM. Пространства имен

Повторим еще раз, что основной идеей, на которой базируется WMI, является представление информации о состоянии любого управляемого объекта в виде стандартной схемы. В качестве такой схемы выступает информационная модель CIM, которая является репозиторием (хранилищем) объектов и классов, моделирующих различные компоненты компьютерной системы.

Таким образом, CIM можно считать хранилищем классов, где класс — это модель (шаблон) управляемого объекта (напомним, что в качестве управляемых объектов могут выступать самые разнообразные логические и физические компоненты компьютерной системы: жесткие диски, журналы событий, сетевые карты, файлы и папки, процессы, сервисы, процессоры и т. д.). С этой точки зрения CIM похожа на другие каталоги, которые используются в Windows (например, каталог файловой системы содержит объекты-файлы и объекты-папки, а каталог Active Directory — объекты-домены, объекты-пользователи, объекты-принтеры и т. д.). Однако важной особенностью CIM является то, что хранящиеся в ней классы чаще всего соответствуют динамически изменяющимся ресурсам, поэтому объекты-экземпляры таких классов не хранятся постоянно в CIM, а создаются провайдером по запросу потребителя WMI. Связано это с тем, что состояние большинства WMI-совместимых устройств меняется очень быстро, и постоянное обновление информации в CIM может значительно снизить общую производительность системы.

> ***Замечание***
>
> Количество классов, имеющихся в CIM, сильно зависит от версии операционной системы. Например, в Windows Server 2003 в CIM "хранится" около 5 000 классов.

Классы, составляющие CIM, имеют свойства и методы и находятся в иерархической зависимости друг от друга — классы-потомки могут наследовать или переопределять свойства родительских классов, а также добавлять собственные свойства. Свойства описывают конфигурацию и текущее состояние управляемого ресурса, а методы позволяют выполнить над этим ресурсом определенные действия.

Классы CIM группируются в *пространства имен* (namespaces), которые упорядочены иерархически (корневое пространство имен обозначается `Root`). *Пространство имен* — это группа логически связанных друг с другом классов, которые относятся к какой-либо определенной технологии или области управления. Например, одно из наиболее часто используемых на практике пространств имен `CIMV2` содержит классы, которые описывают компьютер и операционную систему; некоторые классы из этого пространства имен приведены в табл. П1.2.

Таблица П1.2. *Некоторые классы из пространства имен* `Root\CIMV2`

Класс	Описание
`Win32_BaseBoard`	Описывает системную (материнскую) плату. С помощью экземпляра этого класса можно, например, узнать серийный номер материнской платы
`Win32_Bus`	Описывает физические шины (например, шины PCI или USB) с точки зрения операционной системы Win32
`Win32_Processor`	Представляет процессоры, то есть устройства, способные обрабатывать наборы машинных команд в системе Win32. В мультипроцессорной системе для каждого из процессоров существует отдельный экземпляр этого класса
`Win32_DiskPartition`	Позволяет получать информацию об имеющихся в системе разделах жестких дисков. Для каждого из разделов создается свой экземпляр этого класса
`Win32_FloppyDrive`	Описывает имеющиеся в системе дисководы гибких дисков. Для каждого из дисководов создается свой экземпляр этого класса
`Win32_Keyboard`	Описывает подключенную к компьютеру клавиатуру (например, отражаются количество функциональных клавиш или используемая раскладка)
`Win32_BIOS`	Свойства этого класса представляют атрибуты базовой системы ввода/вывода (BIOS): компания-производитель, версия, номер сборки и т. д.
`Win32_OperatingSystem`	Описывает установленную на компьютере операционную систему. В свойствах экземпляра этого класса хранятся номер сборки системы, используемая по умолчанию кодовая страница, время последней перезагрузки, число пользовательских лицензий и т. п.
`CIM_DataFile`	Экземпляры этого класса соответствуют логическим файлам. Кроме свойств, описывающих различные атрибуты файла, в классе `CIM_DataFile` имеются методы, которые позволяют производить над файлом некоторые действия (копировать, перемещать, переименовывать или удалять его, изменять разрешения на файл и т. д.)
`Win32_Directory`	Экземпляры этого класса соответствуют каталогам файловой системы. Свойства и методы класса `Win32_Directory` практически совпадают со свойствами и методами класса `CIM_DataFile`

Таблица П1.2 (окончание)

Класс	Описание
Win32_Desktop	В свойствах экземпляров класса Win32_Desktop хранятся характеристики рабочих столов пользователей: частота мигания курсора, имя исполняемого файла заставки, частота вызова заставки, имя файла с рисунком рабочего стола и т. д.
Win32_Share	Экземпляры этого класса соответствуют общим ресурсам, имеющимся в системе (общие папки, принтеры, именованные каналы и пр.)
Win32_Service	Экземпляры данного класса позволяют получать информацию о службах (services) операционной системы Win32 и управлять ими (запускать, останавливать и т. п.)
Win32_Process	Каждому запущенному в системе процессу соответствует экземпляр класса Win32_Process. Свойства этого класса позволяют получить полную информацию о процессе (например, его имя, идентификатор, время создания, приоритет), а с помощью методов можно создавать новые процессы, менять приоритеты, завершать процессы и т. д.

Количество и содержимое пространств имен зависит от операционной системы, а также от используемой версии WMI и установленных в системе приложений. Отметим при этом, что в любом варианте установки WMI имеются четыре предопределенных пространства имен, которые всегда находятся на один уровень ниже корневого пространства имен: CIMV2, Default, Security и WMI. Некоторые пространства имен содержат другие пространства. Например, в CIMV2 определены подпространства имен Applications и ms_409.

В операционных системах Windows XP и старше репозиторий CIM физически располагается в четырех файлах каталога %SystemRoot%\System32\Wbem\Repository\FS:

- index.btr (индексный файл);
- objects.data (репозиторий CIM, в котором хранятся описания управляемых ресурсов);
- index.map и object.map (файлы контроля над транзакциями).

Путь к классам и объектам CIM

Все классы внутри одного пространства имен должны иметь уникальные имена (при этом имена классов из разных пространств могут совпадать), причем класс в одном пространстве имен не может иметь предка или потомка из другого пространства. Для идентификации классов и объектов внутри пространства имен в CIM нужно задавать *путь* к этим классам и объектам (object path), аналогично тому, как это делается, например, в пространстве имен файловой системы. Напомним, что любой файл на диске однозначно определяется полным путем к нему следующим образом: указывается имя устройства, одно или несколько имен каталогов и непосредственно имя файла. Другими словами, в файловой системе используется иерархическая структура каталогов и файлов различной степени вложенности. В отличие от этого, у пространств имен CIM имеется только один уровень в глубину, а для идентификации объекта задействуются свойства объектов, которые рассматриваются как ключи. Другими словами, каждый экземпляр класса должен быть однозначно идентифицируемым по своим ключевым параметрам. Полный путь к хранящемуся в CIM классу или объекту-экземпляру класса (управляемому устройству) имеет следующую структуру:

```
[\\ComputerName][\Namespace][:ClassName][.KeyProperty1=Value1
[,KeyProperty2=Value2...]]
```

Здесь *ComputerName* — это сетевое имя компьютера, на котором расположен нужный класс или объект (для задания имени локального компьютера можно использовать символ "."), *Namespace* — название пространства имен, в котором находится этот класс или объект, *ClassName* — имя класса. Параметры *KeyProperty1* и *Value1*, *KeyProperty2* и *Value2*, ... задают список ключевых пар (свойство-значение) объекта. Например, следующий путь

```
\\.\CIMV2:Win32_Process.Name="Notepad.exe"
```

определяет процесс (экземпляр класса `Win32_Process` из пространства имен `CIMV2`) с именем "Notepad.exe", который запущен на локальной машине.

Другой пример — построение пути к объекту WMI, соответствующему десятой записи в журнале событий приложений на компьютере с именем CPU3. В WMI для представления записей в журнале событий строятся объекты-экземпляры класса `Win32_NTLogEvent`. Этот класс хранится в пространстве имен `CIMV2`, а для идентификации конкретного экземпляра класса `Win32_NTLogEvent` используются два ключевых свойства символьного типа: `LogFile` (тип журнала событий) и `RecordNumber` (символьное представление порядкового номера записи в журнале). Поэтому полный путь к объекту в нашем примере будет иметь следующий вид:

```
\\CPU3\CIMV2:Win32_NTLogEvent.LogFile="Application",RecordNumber="10"
```

Безопасность при работе с WMI

В силу своей мощности технология WMI позволяет с помощью специальных утилит или сценариев производить различные потенциально опасные действия (например, остановку служб или перезагрузку компьютера). Причем на удаленной машине такие действия выполнить (или, вернее сказать, попытаться выполнить) так же просто, как и на локальной — достаточно написать имя нужной машины в пути к объекту WMI. Поэтому вопросы безопасности при работе с WMI имеют очень большое значение.

В основном, конечно, технология WMI предназначена для администраторов операционной системы, и вся система безопасности в WMI построена таким образом, чтобы по умолчанию с помощью утилит или сценариев WMI пользователь мог на определенной машине выполнить только те действия, на которые ему даны на ней разрешения. Таким образом реализуется *безопасность WMI на уровне операционной системы;* другими словами, если пользователю на уровне операционной системы не дано, например, право перезагружать компьютер, то и с помощью WMI он не сможет этого сделать.

Дополнительные политики безопасности в WMI реализованы на уровне пространств имен и на уровне протокола DCOM (Distributed COM, распределенная объектная модель компонентов). Перед тем как более подробно рассмотреть эти типы безопасности WMI, напомним основные общие понятия, связанные с безопасностью в Windows.

Безопасность в Windows NT/2000/XP основана на именах пользователей и их паролях. Когда в этих версиях Windows заводится пользователь, то его учетной записи присваивается уникальный *идентификатор безопасности* (Security IDentifier, SID). На основе SID для пользователя формируется *маркер доступа* (Access Token), в который также добавляется список групп, членом которых является пользователь, и список привилегий, которыми он обладает (например, остановка служб или выключение компьютера). Этот маркер доступа присваивается и всем процессам, которые запускает пользователь. В то же время каждый объект операционной системы, доступ к которому определяет система безопасности (это может быть файл, процесс, служба и т. д.), имеет *дескриптор безопасности* (Security Descriptor, SD), в котором хранится *список контроля доступа* (Access Control List, ACL) для этого объекта. При обращении пользователя или запущенного пользователем процесса к объекту происходит сравнение маркера доступа этого пользователя со списком контроля доступа и в зависимости от результатов выдается или отклоняется разрешение на выполнение запрашиваемых действий над объектом.

Механизм *безопасности WMI на уровне пространств имен* соответствует общей модели безопасности Windows. Каждое пространство имен может

иметь собственный дескриптор безопасности, в котором хранится список контроля доступа (ACL). Каждая *запись списка контроля доступа* (Access Control Entry, ACE) содержит информацию о том, какие права (разрешения) имеет определенный пользователь при выполнении различных операций в этом пространстве имен. Список разрешений, используемых при работе с пространством имен, приведен в табл. П1.3.

Таблица П1.3. *Разрешения безопасности для пространства имен WMI*

Разрешение	Описание
Выполнение методов (Execute Methods)	Позволяет вызывать методы классов из определенного пространства имен. Будет ли при этом метод выполнен или же произойдет отказ, зависит от того, имеет ли пользователь разрешение на выполнение этой операции в операционной системе
Полная запись (Full Write)	Разрешает создавать и модифицировать подпространства имен, системные классы и экземпляры классов
Частичная запись (Partial Write)	Открывает возможность создавать и модифицировать любые статические классы и экземпляры несистемных классов
Запись поставщика (Provider Write)	Позволяет записывать в репозиторий CIM классы провайдеров WMI и экземпляры этих классов
Включить учетную запись (Enable Account)	Предоставляет право чтения пространства имен WMI. Пользователи, имеющие это разрешение, могут запускать на локальном компьютере сценарии, которые читают данные WMI
Включить удаленно (Remote Enable)	Разрешает пользователям получить доступ к пространству имен WMI на удаленном компьютере. По умолчанию этим разрешением обладают только администраторы, обычные пользователи не могут получать данные WMI с удаленных машин
Прочесть безопасность (Read Security)	Дает право читать дескриптор безопасности для пространства имен WMI без возможности его модификации
Изменение правил безопасности (Edit Security)	Позволяет изменять дескриптор безопасности для пространства имен WMI

Все записи списка контроля доступа сохраняются в репозитории WMI. Разрешения WMI, определенные для конкретного пространства имен, также применяются ко всем подпространствам имен и классам, которые определены в этом пространстве (наследуются ими). Собственные разрешения безопасности для отдельного класса WMI определить нельзя.

В Windows по умолчанию группа администраторов обладает всеми разрешениями из табл. П1.3, а для остальных пользователей включена учетная запись (Enable Account), разрешено вызывать методы (Execute Methods) и записывать в CIM экземпляры классов провайдеров (Provider Write).

Администратор может изменить разрешения для определенных пользователей с помощью утилиты для настройки параметров WMI (оснастка wmimgmt.msc консоли управления MMC).

Для доступа к инфраструктуре WMI на удаленном компьютере используется коммуникационный протокол DCOM (Distributed COM). При этом пользователь, который запускает сценарий или подключается к WMI с помощью специальных утилит, выступает в качестве клиента, а объект WMI, к которому идет обращение, является сервером. Для того чтобы определить, какой маркер доступа будет применяться при работе с WMI на удаленном компьютере, используются стандартные *уровни олицетворения протокола DCOM* (DCOM Impersonation Levels), которые описаны в табл. П1.4.

Таблица П1.4. *Уровни олицетворения DCOM*

Уровень	Описание
Анонимный доступ (Anonymous)	Объект-сервер не имеет права получить информацию о пользователе или процессе, который обращается к данному объекту (иными словами, объект не может олицетворить клиента). Этот уровень олицетворения в WMI не используется
Идентификация (Identify)	Объект-сервер может запросить маркер доступа, связанный с клиентом, но не может произвести олицетворение. В сценариях WMI этот уровень олицетворения используется редко, так как в этом случае нельзя запускать сценарии WMI на удаленных машинах
Олицетворение (Impersonate)	Объект-сервер может пользоваться всеми правами и привилегиями, которыми обладает клиент. В сценариях WMI рекомендуется использовать именно этот уровень олицетворения, поскольку в таком случае сценарий WMI на удаленной машине сможет выполнять все действия, которые разрешено осуществлять пользователю, запустившему этот сценарий
Делегирование (Delegate)	Объект-сервер, к которому обращается клиент, может обратиться от имени клиента к другому объекту-серверу. Делегирование позволяет сценарию использовать на удаленной машине маркер доступа запустившего его пользователя, а также использовать этот маркер для доступа к объектам WMI на других рабочих станциях. Применение данного уровня олицетворения связано с потенциальным риском, поэтому делегирование в сценариях WMI следует применять только в случае особой необходимости

Выбираемый по умолчанию уровень олицетворения зависит от версии WMI на целевом компьютере. В версиях WMI ниже 1.5 по умолчанию используется уровень Идентификация (Identify), в версии WMI 1.5 и выше — уровень Олицетворение (Impersonate). При необходимости можно изменить уровень олицетворения по умолчанию — для этого необходимо записать наименование нужного уровня (например, `Impersonate` или `Delegate`) в ключ реестра HKEY_LOCAL_MACHINE\Software\Microsoft\Wbem\Scripting\Default Impersonation Level.

Протокол DCOM также предоставляет возможность запросить для соединения WMI определенный уровень аутентификации (проверки подлинности) и конфиденциальности (табл. П1.5).

Таблица П1.5. *Уровни проверки подлинности DCOM*

Уровень	Описание
Отсутствует (None)	Проверка подлинности отсутствует
По умолчанию (Default)	Для выбора уровня проверки подлинности используются стандартные настройки безопасности. Рекомендуется использовать именно этот уровень, так как в таком случае к клиенту будет применен уровень проверки подлинности, который задается сервером
Подключений (Connect)	Клиент проходит проверку подлинности только во время подключения к серверу. После того как соединение установлено, никаких дополнительных проверок не производится
Вызовов (Call)	Клиент проходит проверку подлинности в начале каждого вызова во время приема запросов сервером. При этом заголовки пакетов подписываются, однако сами данные (содержимое пакетов), передаваемые между клиентом и сервером, не подписываются и не шифруются
Пакетов (Pkt)	Проверке подлинности подвергаются все пакеты данных, которые поступают серверу от клиентов. Как и при проверке подлинности на уровне вызовов, заголовки пакетов подписываются, но не шифруются. Сами пакеты не подписываются и не шифруются
Целостности пакетов (PktIntegrity)	Все пакеты данных проходят проверку подлинности и целостности, то есть проверяется, что содержимое пакета не было изменено во время передачи от клиента серверу. При этом данные подписываются, но не шифруются
Секретности пакетов (PktPrivacy)	Все пакеты данных проходят проверку подлинности и целостности, при этом данные подписываются и шифруются, что обеспечивает конфиденциальность передаваемых данных

Отметим, что DCOM может и не установить запрошенный уровень проверки подлинности. Например, при локальной работе с WMI всегда используется уровень секретности пакетов (PktPrivacy).

Структура классов WMI

Напомним, что всякому ресурсу, управляемому с помощью WMI, соответствует специальный класс WMI; каждый класс имеет четко определенную структуру и содержит *свойства*, *методы* и *квалификаторы* (свои квалификаторы могут быть также у свойств и методов). Классы описываются с помощью специального языка MOF (Managed Object Format), который, в свою очередь, базируется на языке IDL (Interface Definition Language), применяемом для описания интерфейсов COM-объектов. После определения структуры класса с помощью MOF разработчик может добавить откомпилированное представление этого класса в репозиторий CIM с помощью стандартной утилиты mofcomp.exe.

Подробнее понятия свойств, методов и квалификаторов будут обсуждаться далее, а начнем мы с типизации классов CIM.

Основные типы классов CIM

В CIM существует три основных типа классов, различающихся между собой по способу хранения информации об управляемых ресурсах:

- *абстрактный класс* (abstract class) представляет собой шаблон, который служит исключительно для образования новых классов-потомков (абстрактных и неабстрактных). Абстрактный класс не может непосредственно использоваться для получения экземпляра управляемого ресурса;
- *статический класс* (static class) определяет данные, которые физически хранятся в репозитории CIM (к такому типу относятся, например, данные о собственных настройках WMI), вследствие чего для доступа к экземплярам статических классов не нужно прибегать к помощи каких-либо провайдеров;
- *динамический класс* (dynamic class) моделирует управляемый ресурс, данные о котором соответствующий провайдер возвращает в динамическом режиме.

Кроме трех основных типов классов в CIM выделяется еще один специальный тип — *ассоциативный класс* (association class) — это абстрактный, статический или динамический класс, который описывает логическую связь между двумя классами или управляемыми ресурсами. Например, ассоциативный класс `Win32_SystemProcesses` связывает класс `Win32_Process`, экзем-

пляры которого соответствуют запущенным в системе процессам, с классом `Win32_ComputerSystem`, в котором представлены общие настройки компьютерной системы.

Кроме этого, все классы CIM можно разделить на четыре группы по принадлежности к различным информационным моделям.

- **Системные классы.** Системными называются те классы, которые служат для задания конфигурации и выполнения внутренних функций WMI (определение пространств имен, обеспечение безопасности при работе с пространствами имен, регистрация провайдеров, подписка на события WMI и формирование сообщений о наступлении таких событий). Системные классы могут быть абстрактными или статическими. Системные классы можно легко отличить от других по названию: имена всех системных классов начинаются с символов "__" (двойное подчеркивание), например, `__SystemClass`, `__NAMESPACE`, `__Provider` или `__Win32Provider`.
- **Классы модели ядра (основной модели)** (core model). К этой группе относятся абстрактные классы, которые обеспечивают интерфейс со всеми областями управления. Названия таких классов начинаются с префикса "`CIM_`". Примерами классов модели ядра могут служить класс `CIM_ManagedSystemElement` (свойства этого класса идентифицируют управляемые компоненты системы) и его наследники `CIM_LogicalElement` (описание логического управляемого ресурса, например, файла или каталога) и `CIM_PhysicalElement` (описание физического управляемого ресурса, например, периферийного устройства).
- **Классы общей модели** (common model). Общая модель является расширением основной модели — здесь представлены классы, которые являются специфическими для задач управления, но не зависят от конкретной технологии или реализации (другими словами, не зависят от типа операционной системы). Названия таких классов, как и классов модели ядра, начинаются с "`CIM_`". Класс `CIM_LogicalFile` (наследник класса `CIM_LogicalElement`), описывающий файл, является примером класса общей модели, так как файловая система присутствует практически в любой операционной системе.
- **Классы модели расширения** (extension model). Эта категория классов включает в себя специфические для каждой технологии или реализации дополнения к общей модели. В WMI определено большое количество классов, которые соответствуют ресурсам, специфическим для среды Win32 (имена этих классов начинаются с префикса "`Win32_`"). Например, классы `Win32_PageFile` и `Win32_ShortCutFile`, которые описывают, соответственно, файлы подкачки Windows и файлы-ярлыки, являются потомками класса `CIM_LogicalFile` из общей модели.

Свойства классов WMI

Свойства классов используются для однозначной идентификации экземпляра класса, представляющего конкретный управляемый ресурс, а также для описания текущего состояния этого ресурса. Рассмотрим два простых примера — классы из пространства имен `CIMV2`, являющиеся шаблонами для служб и процессов Windows.

Напомним, что для просмотра списка всех служб, установленных на компьютере, можно воспользоваться оснасткой **Службы** консоли управления MMC (рис. П1.2).

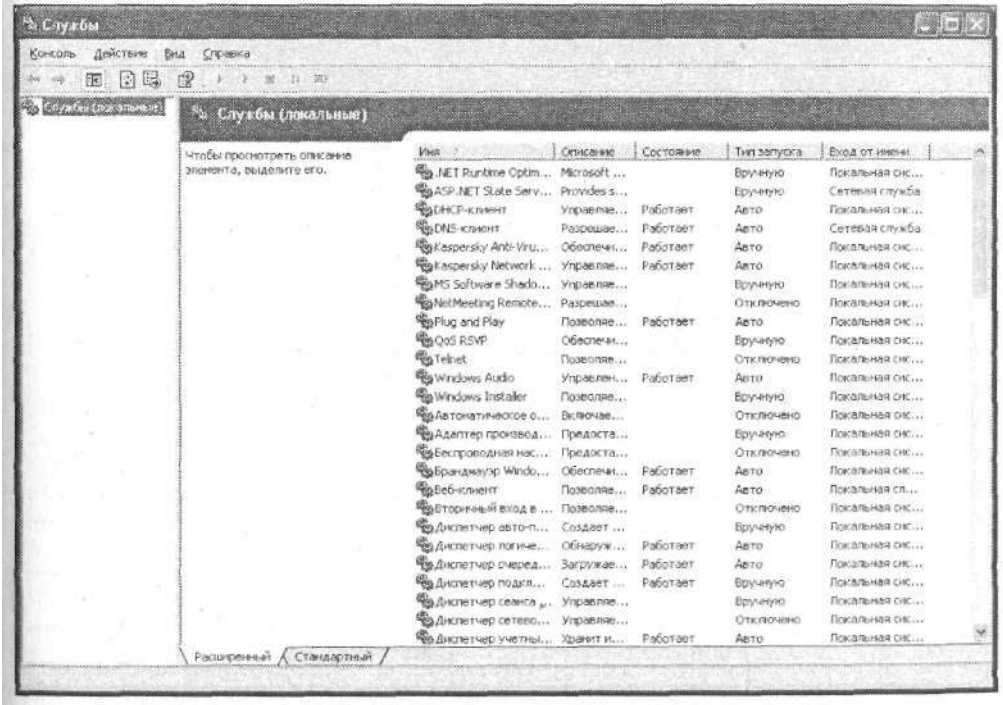


Рис. П1.2. Службы, зарегистрированные на локальном компьютере

Параметры определенной службы можно просматривать и изменять с помощью диалогового окна с несколькими вкладками, которое появляется после двойного щелчка по соответствующей службе строке в правом окне консоли (рис. П1.3).

Службам Windows в WMI соответствуют экземпляры класса `Win32_Service`; основные свойства этого класса приведены в табл. П1.6.

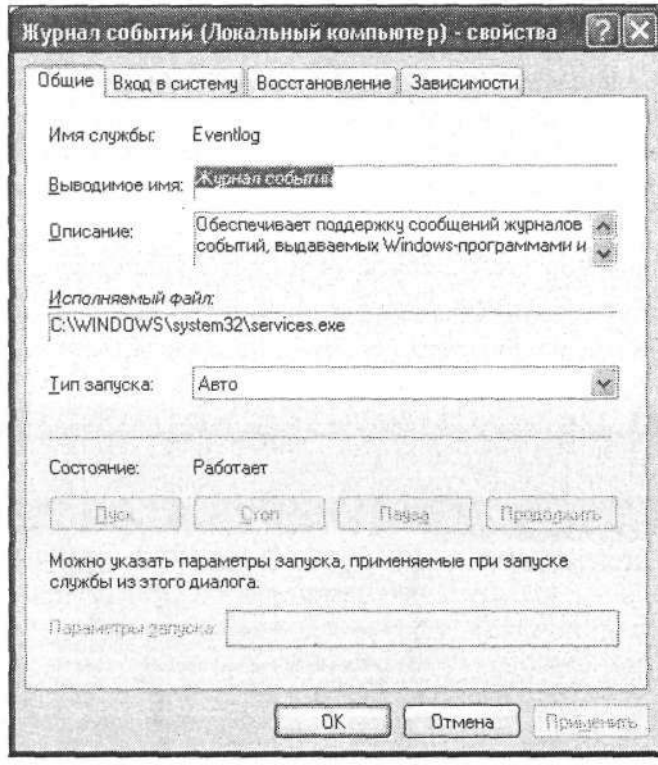


Рис. П1.3. *Параметры службы* ***Журнал событий***

Таблица П1.6. *Некоторые свойства класса* `Win32_Service`

Свойство	Описание
`AcceptPause`	Свойство логического типа, значение которого равно `True`, если службу можно приостановить, и равно `False` в противном случае
`AcceptStop`	Свойство логического типа, значение которого равно `True`, если службу можно остановить, и равно `False` в противном случае
`Caption`	Краткое описание службы
`Description`	Полное описание службы
`DesktopInteract`	Свойство логического типа, значение которого равно `True`, если служба может взаимодействовать с рабочим столом пользователей, и равно `False` в противном случае
`DisplayName`	Имя службы, которое выводится в списке служб

Таблица П1.6 (окончание)

Свойство	Описание
ErrorControl	Строка, задающая действие программы загрузки, которое будет выполнено в случае возникновения сбоя при запуске службы во время загрузки операционной системы: Ignore — пользователю не будет выведено никаких сообщений о сбое, Normal — будет выведено сообщение о сбое при запуске службы, Critical — система попытается автоматически произвести перезагрузку в хорошей конфигурации, Unknown — действие для подобного типа ошибок не определено
Name	Имя службы
PathName	Полный путь к бинарному файлу, соответствующему службе
ProcessId	Уникальный идентификатор службы
ServiceType	Строка, описывающая тип службы: Kernel Driver, File System Driver, Adapter, Recognizer Driver, Own Process, Share Process, Interactive Process
Started	Свойство логического типа, значение которого равно True, если служба была запущена, и равно False в противном случае
StartMode	Строка, описывающая способ загрузки службы: Boot (применяется только при загрузке служб для драйверов), System (применяется только при загрузке служб для драйверов), Auto (служба загружается автоматически), Manual (служба может быть запущена вручную), Disabled (службу запустить нельзя)
StartName	Учетная запись, от имени которой запускается служба
State	Текущее состояние службы: Stopped (остановлена), Start Pending (стартует), Stop Pending (останавливается), Running (запущена), Continue Pending (возвращается в активное состояние), Pause Pending (приостанавливается), Paused (приостановлена), Unknown (состояние службы определить не удалось)
WaitHint	Примерное время (в миллисекундах), необходимое для выполнения операций приостановки, остановки или запуска службы

Как мы видим, многие свойства класса Win32_Service соответствуют элементам ввода в диалоговом окне свойств службы. Например, для экземпляра класса Win32_Service, который представляет службу **Журнал событий**, свойства Name, StartMode, DisplayName и Description равны соответственно "Eventlog", "Auto" (что соответствует русскому "Авто"), "Журнал событий" и "Обеспечивает поддержку сообщений журналов событий, выдаваемых Windows-программами и компонентами системы, и просмотр этих сообщений. Эта служба не может быть остановлена." (см. рис. П1.3).

Рис. П1.4. Процессы, запущенные на локальном компьютере

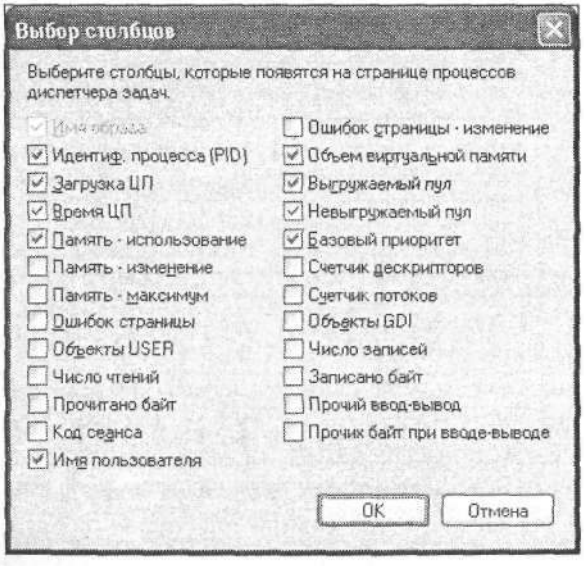


Рис. П1.5. Выбор столбцов для отображения в Диспетчере задач

Рассмотрим теперь класс `Win32_Process`, экземпляры которого соответствуют запущенным в операционной системе процессам. Напомним, что информацию обо всех процессах можно получить с помощью Диспетчера задач, запускаемого нажатием клавиш <Ctrl>+<Shift>+<Esc> (рис. П1.4).

Количество выводимых на экран параметров процессов зависит от настроек Диспетчера задач: выбрав в меню **Вид** пункт **Выбрать столбцы**, можно отметить интересующие нас параметры (рис. П1.5).

Параметрам запущенного процесса соответствуют свойства класса `Win32_Process`; некоторые из этих свойств приведены в табл. П1.7.

Таблица П1.7. *Некоторые свойства класса* `Win32_Process`

Свойство	Описание
`Caption`	Короткое текстовое описание процесса
`CommandLine`	Командная строка, при помощи которой процесс был запущен
`CreationDate`	Время начала выполнения процесса
`Description`	Полное описание процесса
`ExecutablePath`	Полный путь к исполняемому файлу процесса
`HandleCount`	Общее количество дескрипторов, открытых в настоящее время процессом (равно общему количеству дескрипторов, открытых каждым потоком в процессе)
`MaximumWorkingSetSize`	Максимально возможный размер рабочего набора процесса (*рабочий набор процесса* — это набор страниц, доступных процессу в физической оперативной памяти), в байтах
`MinimumWorkingSetSize`	Минимально возможный размер рабочего набора процесса, в байтах
`Name`	Имя процесса
`OtherOperationCount`	Число выполненных операций ввода/вывода, отличных от операции чтения или записи
`OtherTransferCount`	Размер данных, переданных в процессе выполнения операций, отличных от операции чтения или записи, в байтах
`PageFileUsage`	Размер части файла подкачки, которая используется процессом в настоящее время, в байтах
`ParentProcessID`	Уникальный идентификатор родительского процесса, создавшего данный процесс

Таблица П1.7 *(окончание)*

Свойство	Описание
PeakPageFileUsage	Максимальный размер части файла подкачки, которая использовалась процессом за все время его работы, в байтах
PeakVirtualSize	Максимальное значение размера виртуального адресного пространства, которое использовалось процессом единовременно, в байтах
PeakWorkingSetSize	Максимальное значение размера рабочего набора процесса за все время работы, в байтах
Priority	Приоритет процесса (минимальному приоритету соответствует значение 0, максимальному — 31)
ProcessID	Уникальный идентификатор процесса. Значение этого свойства актуально с момента создания процесса до окончания его работы
ReadOperationCount	Число выполненных процессом операций чтения
ReadTransferCount	Размер прочитанных данных, в байтах
ThreadCount	Число активных потоков в процессе
VirtualSize	Текущий размер виртуального адресного пространства, используемого процессом, в байтах
WorkingSetSize	Размер памяти, необходимый для успешного выполнения процесса в операционной системе, использующей страничную организацию памяти, в байтах
WriteOperationCount	Число выполненных процессом операций записи
WriteTransferCount	Размер записанных данных, в байтах

Отметим, что в основном в WMI свойства классов доступны только для чтения, однако значения определенных свойств в экземплярах некоторых классов можно изменять напрямую (для этого применяется специальный метод `Put_()`). Например, в экземплярах класса `Win32_LogicalDisk`, которые соответствуют логическим дискам, можно изменять свойство `VolumeName`, где хранится метка соответствующего диска.

> ***Замечание***
>
> Количество свойств, значения которых можно изменять, зависит от операционной системы. Например, в Windows 2000 для записи доступны только 39 свойств, а в Windows XP — 145 свойств.

Для того чтобы узнать, является ли определенное свойство доступным для записи, нужно проверить значение квалификатора `Write` этого свойства.

Методы классов WMI

Методы класса позволяют выполнять те или иные действия над управляемым ресурсом, которому соответствует этот класс (так как не над каждым ресурсом можно производить какие-либо операции, то не у всякого класса есть методы). Для примера в табл. П1.8 и П1.9 описаны методы, которые имеются у классов `Win32_Service` (службы Windows) и `Win32_Process` (процессы Windows).

Таблица П1.8. *Методы класса* `Win32_Service`

Метод	Описание
`StartService()`	Запускает службу
`StopService()`	Останавливает службу
`PauseService()`	Приостанавливает службу
`ResumeService()`	Возобновляет работу службы
`UserControlService(n)`	Посылает службе заданный пользователем код *n* (число от 128 до 255)
`Create(`*Name, DisplayName, PathName, ServiceType, ErrorControl, StartMode, DesktopInteract, StartName, StartPassword, LoadOrderGroup, LoadOrderGroupDependencies, ServiceDependencies*`)`	Создает службу
`Change(`*DisplayName, PathName, ServiceType, ErrorControl, StartMode, DesktopInteract, StartName, StartPassword, LoadOrderGroup, LoadOrderGroupDependencies, ServiceDependencies*`)`	Изменяет параметры службы
`ChangeStartMode(`*StartMode*`)`	Изменяет тип загрузки службы. Символьный параметр *StartMode* может принимать следующие значения: "Boot" (применяется только при загрузке служб для драйверов), "System" (применяется только при загрузке служб для драйверов), "Auto" (служба загружается автоматически), "Manual" (служба может быть запущена вручную), "Disabled" (службу запустить нельзя)
`Delete()`	Удаляет существующую службу

Таким образом, методы класса `Win32_Service` позволяют изменять значения некоторых свойств этого класса и выполнять манипуляции над конкретной службой: запускать ее, приостанавливать, останавливать и т. д.

Перейдем к рассмотрению методов класса `Win32_Process`, которые описаны в табл. П1.9.

Таблица П1.9. *Методы класса* `Win32_Process`

Метод	Описание
`AttachDebugger()`	Запускает отладчик, установленный в системе по умолчанию, для отладки процесса
`Create(`*`CommandLine`*`,` *`CurrentDirectory`*`,` *`ProcessStartupInformation`*`,` *`ProcessId`*`)`	Создает новый неинтерактивный процесс
`GetOwner(`*`User`*`,` *`Domain`*`)`	Извлекает данные о процессе: после выполнения этого метода в переменной *`User`* будет записано имя пользователя, создавшего процесс (владельца процесса), а в переменной *`Domain`* — имя домена, в котором запущен этот процесс
`GetOwnerSid(`*`Sid`*`)`	Возвращает в переменной *`Sid`* идентификатор безопасности (Security IDentifier, SID) владельца процесса
`SetPriority(`*`Priority`*`)`	Устанавливает приоритет процесса. Числовой параметр *`Priority`* определяет требуемый приоритет и может принимать следующие значения: 64 (низкий), 16 384 (ниже среднего), 32 (средний), 32 768 (выше среднего), 128 (высокий), 256 (процесс выполняется в реальном времени)
`Terminate(`*`Reason`*`)`	Завершает процесс и все его потоки. Числовой параметр *`Reason`* задает код выхода, который будет сообщен операционной системе после завершения процесса

Таким образом, методы класса `Win32_Process` позволяют выполнять над процессами все те действия, которые можно осуществить в **Диспетчере задач Windows** с помощью контекстного меню, появляющегося после щелчка правой кнопкой мыши над выделенным процессом в списке (см. рис. П1.6), и кнопки **Завершить процесс** (Terminate process), а также ряд других действий.

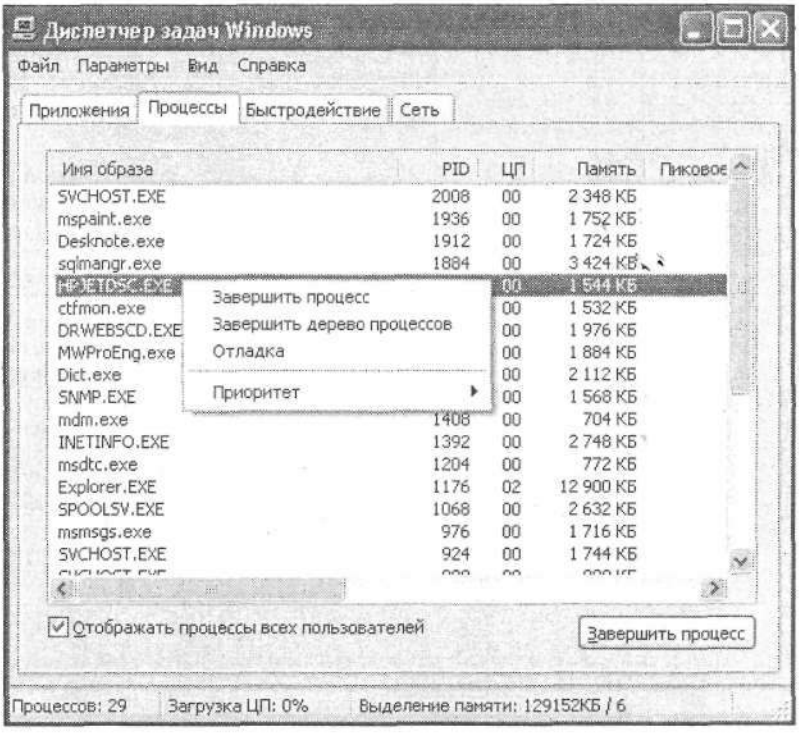


Рис. П1.6. Контекстное меню, позволяющее выполнять действия *над выделенным процессом*

Квалификаторы классов, свойств и методов

В WMI для классов, свойств и методов можно задать так называемые *квалификаторы* (qualifiers). Квалификаторы содержат дополнительную информацию о том классе, свойстве или методе, в котором они определены.

Квалификаторы классов

Квалификаторы классов предоставляют информацию о классе в целом. Например, тип класса описывает квалификаторы логического типа `CIM_BOOLEAN` с именами `Abstract` (абстрактный класс), `Dynamic` (динамический класс) и `Association` (ассоциативный класс).

Один и тот же класс в различных операционных системах может иметь разное количество квалификаторов (версия WMI, поставляемая с Windows XP, соответствует спецификации CIM 2.5, а версии WMI в Windows 2000 и ниже — спецификации CIM 2.0). Для примера в табл. П1.10 приведено описание квалификаторов для класса `Win32_Service` в Windows XP.

Таблица П1.10. *Квалификаторы класса* `Win32_Service`

Квалификатор	Тип	Значение	Описание
`Dynamic`	`CIM_BOOLEAN`	`True`	Тип класса
`Locale`	`CIM_SINT32`	`1033`	Язык по умолчанию для класса или экземпляра класса
`Provider`	`CIM_STRING`	`CIMWin32`	Имя провайдера класса
`SupportsUpdate`	`CIM_BOOLEAN`	`True`	Указывает на то, что класс поддерживает операцию изменения (обновления) экземпляров
`UUID`	`CIM_STRING`	`{8502C4D9-5FBB-11D2-AAC1-006008C78BC7}`	Универсальный уникальный идентификатор класса

Класс `Win32_Process` позволяет создавать новые процессы и завершать уже существующие, поэтому в данном классе появляется несколько новых квалификаторов (табл. П1.11).

Таблица П1.11. *Квалификаторы класса* `Win32_Process`

Квалификатор	Тип	Значение	Описание
`CreateBy`	`CIM_STRING`	`Create`	Название метода, при помощи которого создается экземпляр класса
`DeleteBy`	`CIM_STRING`	`DeleteInstance`	Название метода, при помощи которого уничтожается экземпляр класса
`Dynamic`	`CIM_BOOLEAN`	`True`	Тип класса
`Locale`	`CIM_SINT32`	`1033`	Язык по умолчанию для класса или экземпляра класса
`Provider`	`CIM_STRING`	`CIMWin32`	Имя провайдера класса
`SupportsCreate`	`CIM_BOOLEAN`	`True`	Указывает на то, что класс поддерживает операцию создания экземпляров

Таблица П1.11 (окончание)

Квалификатор	Тип	Значение	Описание
SupportsDelete	CIM_BOOLEAN	True	Указывает на то, что класс поддерживает операцию уничтожения экземпляров
UUID	CIM_STRING	{8502C4DC-5FBB-11D2-AAC1-006008C78BC7}	Универсальный уникальный идентификатор класса

Квалификаторы свойств

Квалификаторы свойств позволяют определить тип данного свойства (квалификатор CIMType), доступность его для чтения (квалификатор Read) и записи (квалификатор Write) и т. п. Для примера в табл. П1.12 приведено описание квалификаторов свойства ServiceType класса Win32_Service (напомним, что это свойство описывает тип службы).

Таблица П1.12. *Квалификаторы свойства* ServiceType *класса* Win32_Service

Квалификатор	Тип	Значение	Описание
CIMType	CIM_STRING	String	Тип свойства
MappingStrings	CIM_STRING | CIM_FLAG_ARRAY	Win32API|Service Structures|QUERY_SERVICE_CONFIG|dwServiceType	Множество значений (ключевых слов), по которым можно найти дополнительную информацию о данном свойстве
Read	CIM_BOOLEAN	True	Указывает на то, что свойство доступно для чтения
ValueMap	CIM_STRING | CIM_FLAG_ARRAY	Kernel Driver, File System Driver, Adapter, Recognizer Driver, Own Process, Share Process, Interactive Process	Набор допустимых значений для свойства

Замечание

Напрямую с помощью метода Put_() можно изменять значения только тех свойств, у которых имеется квалификатор Write со значением True.

Квалификаторы методов

Квалификаторы методов могут описывать множество допустимых значений, которые будут возвращаться методом (квалификатор `ValueMap`), указывать права, которыми необходимо обладать для вызова метода (квалификатор `Privileges`) и т. п. Для примера в табл. П1.13 приведено описание квалификаторов метода `Create` класса `Win32_Process` (напомним, что этот метод используется для запуска в системе нового процесса).

Таблица П1.13. *Квалификаторы метода* `Create` *класса* `Win32_Process`

Квалификатор	Тип	Значение	Описание
`Constructor`	`CIM_BOOLEAN`	`True`	Указывает на то, что данный метод используется для создания экземпляров класса
`Implemented`	`CIM_BOOLEAN`	`True`	Указывает на то, что данный метод реализован в провайдере
`MappingStrings`	`CIM_STRING | CIM_FLAG_ARRAY`	`Win32API| Process and Thread Functions| CreateProcess`	Множество значений (ключевых слов), по которым можно найти дополнительную информацию о данном методе
`Privileges`	`CIM_STRING | CIM_FLAG_ARRAY`	`SeAssignPrimaryTokenPrivilege, SeIncreaseQuotaPrivilege`	Перечисляет права, необходимые для выполнения данного метода
`Static`	`CIM_BOOLEAN`	`True`	Указывает на то, что данный метод статический, то есть не может быть вызван из экземпляра класса
`ValueMap`	`CIM_STRING | CIM_FLAG_ARRAY`	`0, 2, 3, 8, 9, 21, ..`	Набор возвращаемых данным методом значений

Замечание

Выполнять можно только те методы, у которых имеется квалификатор `Implemented` со значением `True`.

Интерактивная работа с объектами WMI

При изучении WMI очень полезно с помощью графических утилит просматривать иерархическую структуру классов и связи между различными объектами CIM. Для просмотра объектной модели WMI можно воспользоваться специальным тестером WMI (стандартная программа в операционной системе Windows) или утилитами из разработанного Microsoft дополнительного пакета WMI Tools.

Тестер WMI (WBEMTest)

Тестер WMI (wbemtest.exe) — это графическая утилита, с помощью которой можно взаимодействовать с инфраструктурой WMI на локальном или удаленном компьютере. С помощью тестера WMI можно решать следующие задачи:

- подсоединяться к определенному пространству имен CIM;
- создавать и удалять классы и экземпляры классов;
- получать список имеющихся классов и экземпляров классов CIM;
- просматривать и изменять свойства и квалификаторы классов или экземпляров классов;
- выполнять методы классов и экземпляров классов;
- составлять и выполнять запросы на языке WQL;
- выводить код MOF для классов и экземпляров управляемых ресурсов.

Исполняемый файл wbemtest.exe является стандартным компонентом WMI в любой версии операционной системы Windows; устанавливается он в каталог %SystemRoot%\System32\Wbem. После запуска этого файла появляется диалоговое окно **Тестер инструментария управления Windows**, с помощью которого можно получить доступ ко всем функциям тестера WMI (рис. П1.7).

Сразу после запуска большинство кнопок этого диалогового окна недоступны — ими можно будет воспользоваться только после подключения к подсистеме WMI. До подключения можно установить флажок **Включить все привилегии**, что позволит средствами WMI выполнять операции, для которых необходимы специальные привилегии в операционных системах Windows (например, перезагрузку компьютера).

Работа с тестером WMI предполагает хорошее знание структуры CIM и умение составлять запросы на языке WQL. Для первоначального ознакомления и изучения структуры объектной модели WMI лучше воспользоваться пакетом WMI Tools.

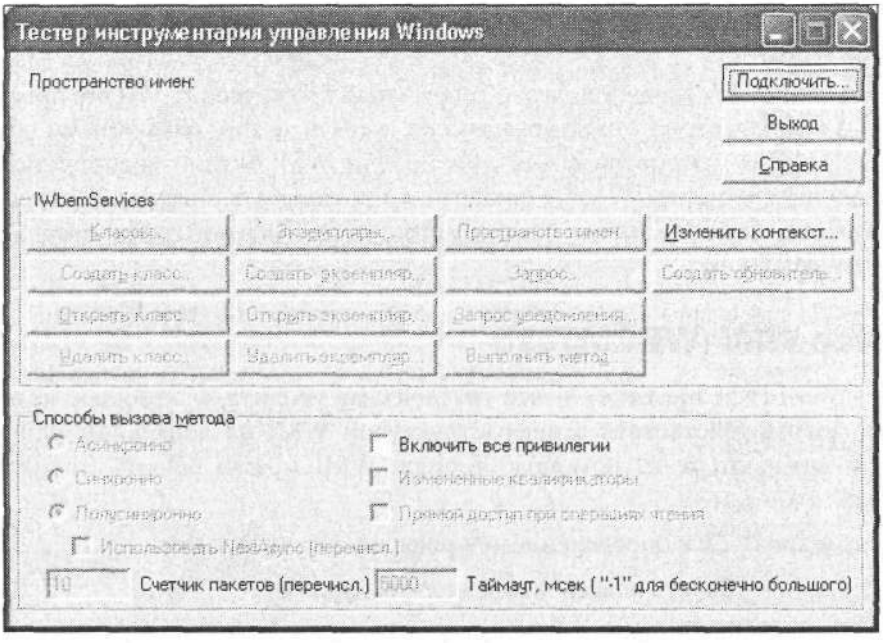


Рис. П1.7. Тестер WMI

Административные утилиты WMI (WMI Tools)

В состав разработанных Microsoft административных утилит WMI входят несколько приложений, имеющих сходный интерфейс.

- **WMI CIM Studio.** Это наиболее универсальное приложение, которое может быть использовано для просмотра и редактирования классов и их экземпляров в репозитории CIM. С помощью WMI CIM Studio можно также выполнять методы классов и объектов, просматривать ассоциации между различными классами, выполнять запросы на языке WQL, генерировать и компилировать файлы MOF для классов и объектов. Короче говоря, утилита WMI CIM Studio обладает практически теми же возможностями, что и тестер WMI (wbemtest.exe), однако имеет гораздо более удобный интуитивный интерфейс. Как и работа с тестером WMI, использование WMI CIM Studio предполагает довольно хорошее знание структуры репозитория CIM и названий нужных классов.
- **WMI Object Browser.** Эта утилита предназначена для просмотра и редактирования объектов (экземпляров классов) в репозитории CIM, а также для вызова их методов. Особенностью WMI Object Browser является то, что информация об объектах представлена в виде иерархического дерева,

где в качестве корневого объекта может использоваться произвольный экземпляр выбранного нами класса. Само дерево объектов строится с помощью ассоциированных классов, что помогает извлекать информацию об управляемых ресурсах, не обладая глубокими знаниями о структуре репозитория CIM и используемых классах.

- **WMI Event Registration Tool.** Данная утилита предоставляет графический интерфейс для регистрации и конфигурирования постоянных потребителей событий WMI. Здесь можно создавать или изменять фильтры событий, определять постоянных потребителей и устанавливать связи между ними и фильтрами событий.
- **WMI Event Viewer.** Это вспомогательное приложение является постоянным потребителем событий, позволяющим сортировать и просматривать подробную информацию о полученных событиях.

Инсталляционный файл WMITools.exe можно свободно скачать с сервера Microsoft (**http://download.microsoft.com/download/.NetStandardServer/Install/V1.1/NT5XP/EN-US/WMITools.exe**).

Административные утилиты WMI реализованы в виде элементов ActiveX, которые встроены в страницы HTML, поэтому для их корректной работы необходимо, чтобы политика безопасности веб-браузера не блокировала работу этих элементов. Кроме этого, пользователь, который производит установку утилит WMI на компьютер, должен обладать правами администратора.

Подключение к пространству имен WMI

Для запуска любой из трех основных административных утилит WMI (WMI CIM Studio, WMI Object Browser или WMI Event Registration Tool) нужно выбрать соответствующий одноименный пункт в меню **Пуск | Программы | WMI Tools**. Первым шагом после запуска во всех этих утилитах является подключение к какому-либо пространству имен на локальном или удаленном компьютере с помощью диалогового окна **Connect to namespace** (рис. П1.8).

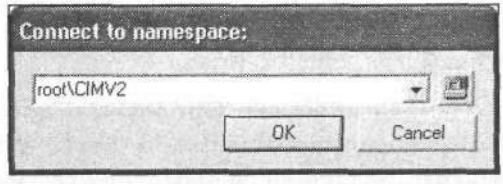


Рис. П1.8. Ввод пространства имен WMI для подключения

По умолчанию в этом окне предлагается подключиться к пространству имен `CIMV2` на локальном компьютере (`root\CIMV2`). Путь к нужному пространству имен на локальном или удаленном компьютере можно либо написать вручную, либо выбрать это пространство с помощью кнопки **Browse For Namespace**.

Если компьютер, к которому предполагается подключиться, доступен в сети, то его можно найти и выбрать с помощью кнопки **Network Neighborhood**.

После ввода в поле **Machine Name** имени нужного компьютера следует выбрать интересующее нас пространство имен из списка. Для этого нужно в поле **Starting Namespace** написать название корневого пространства имен (обычно это `Root`) и нажать кнопку **Connect**. В результате на экран выводится диалоговое окно **Login** (в заголовке этого окна отображается также название запущенного приложения), в котором можно указать имя пользователя и пароль для учетной записи, от имени которой происходит подключение к пространству имен (рис. П1.9).

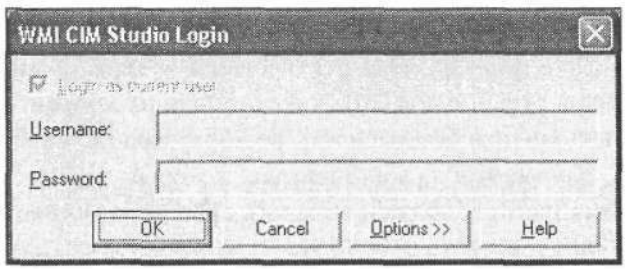


Рис. П1.9. Диалоговое окно **WMI CIM Studio Login**

Напомним, что доступ к пространству имен на локальном компьютере может получить только текущий пользователь, поэтому в случае локальной машины флажок **Login as current user** установлен, а поля **Username**, **Password** и **Authority** недоступны для редактирования. К пространству имен на удаленной машине можно подключаться как от имени текущего пользователя (для этого следует установить флажок **Login as current user**), так и от имени другого пользователя (в этом случае нужно снять флажок **Login as current user** и внести в поля **Username** и **Password** имя пользователя в виде *`domain\user`* и пароль, соответственно).

Также в окне **Login** с помощью кнопки **Options>>** можно настроить дополнительные параметры подключения к инфраструктуре WMI (задать уровни олицетворения и проверки подлинности протокола DCOM, а также указать требуемые привилегии операционной системы).

После нажатия кнопки **OK** в окне **Login** программа подключится к структуре WMI на нужном компьютере, и в окне **Browse For Namespace** появится иерархический список всех классов из репозитория CIM на этом компьютере (рис. П1.10).

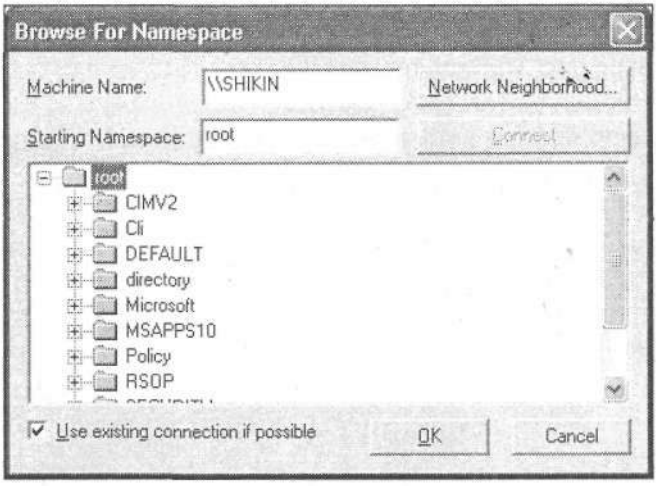


Рис. П1.10. Выбор нужного пространства имен из списка

Выбрав требуемое пространство имен из списка и нажав кнопку **OK**, мы вновь попадаем в диалоговое окно **Login** (см. рис. П1.9), где нужно ввести необходимую информацию и нажать кнопку **OK**.

> ***Замечание***
>
> Независимо от способа подключения пользователь должен иметь доступ к выбранному пространству имен WMI.

Для изучения структуры классов WMI можно воспользоваться утилитой WMI CIM Studio или WMI Object Browser.

WMI CIM Studio

Утилита WMI CIM Studio является универсальным инструментом при работе со схемой CIM. Она позволяет:

- осуществлять навигацию по иерархическому дереву классов CIM;
- формировать список всех экземпляров определенного класса;
- добавлять новые и удалять существующие классы или экземпляры классов (объекты);

- просматривать и изменять (если это возможно) свойства, методы, квалификаторы и ассоциации классов или объектов;
- выполнять методы классов или объектов;
- генерировать для класса или объекта его описание на языке MOF и компилировать имеющийся MOF-файл в репозиторий CIM.

Приложение WMI CIM Studio реализовано в виде двух окон, которые открываются в браузере Internet Explorer (рис. П1.11).

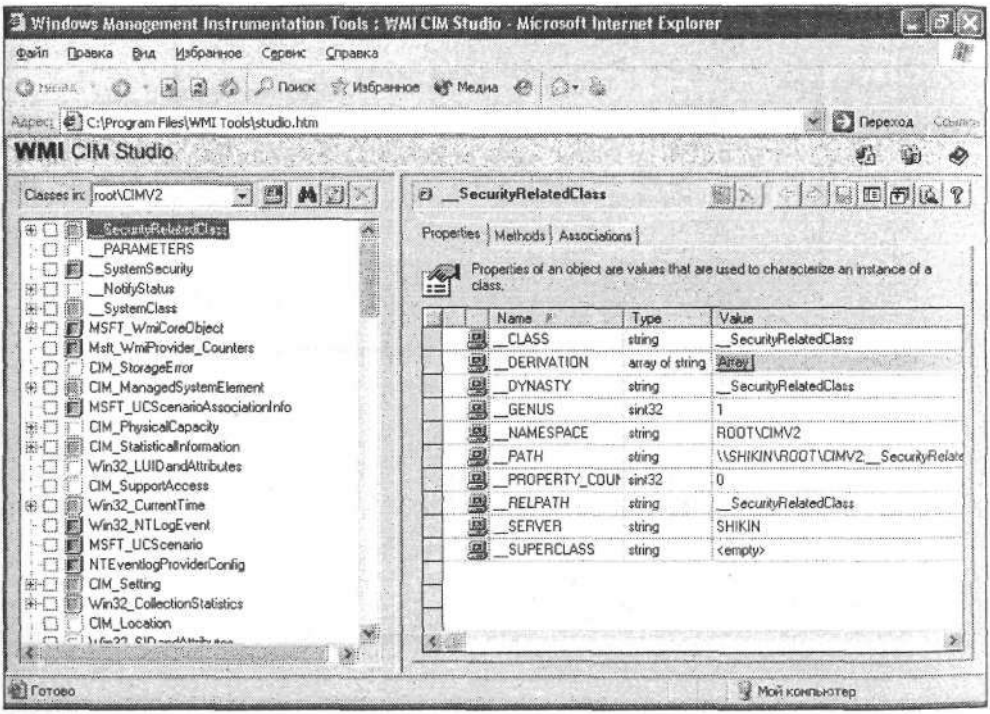


Рис. П1.11. Главное окно утилиты WMI CIM Studio

Левое окно называется *проводником классов* (Class Explorer), а правое — *просмотрщиком классов* (Class Viewer). Выбрав класс в левом окне, можно просмотреть информацию о нем в правом окне.

WMI Object Browser

Утилита WMI Object Browser позволяет осуществлять навигацию по иерархическому дереву объектов WMI, просматривать и редактировать (если это возможно) свойства, методы, квалификаторы и ассоциации экземпляров

классов, а также выполнять методы этих экземпляров. Главное отличие WMI Object Browser от WMI CIM Studio состоит в том, что в WMI CIM Studio мы пользуемся списком классов CIM в выбранном пространстве имен WMI, а в WMI Object Browser на экран выводится дерево объектов, причем в качестве корневого объекта здесь может использоваться произвольный экземпляр выбранного нами класса, а само дерево объектов строится с помощью ассоциативных классов.

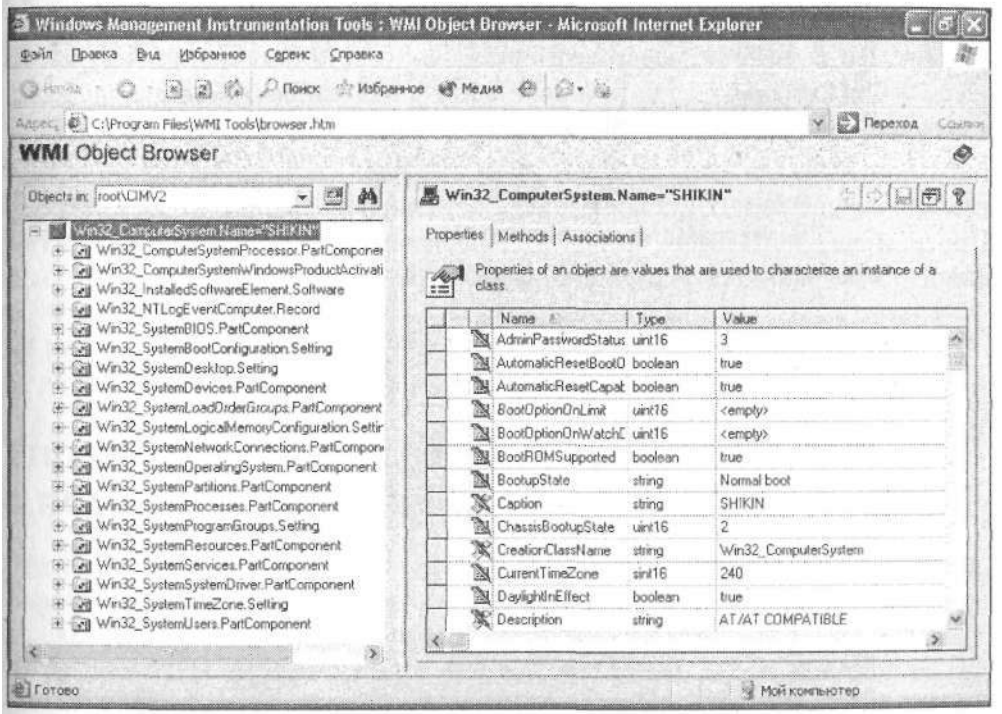


Рис. П1.12. Утилита WMI Object Browser

Внешне утилита WMI Object Browser очень похожа на WMI CIM Studio, здесь также имеются два окна, которые открываются в браузере Internet Explorer (рис. П1.12). Левое окно называется *проводником объектов* (Object Explorer), а правое — *просмотрщиком объектов* (Object Viewer). После выбора объекта в левом окне информация о нем будет показана в правом.

Можно сказать, что утилита WMI Object Browser разработана, в большей мере, для применения системными администраторами, которые могут не иметь детального представления о назначении конкретных классов WMI и о структуре CIM вообще. Здесь вся информация (схема), которая содержится в репо-

зитории CIM, представлена в более понятном и наглядном виде, чем в WMI CIM Studio. Например, по умолчанию корневым объектом в пространстве имен `CIMV2` на локальном или удаленном компьютере является экземпляр класса `Win32_ComputerSystem`, у которого значение свойства `Name` совпадает с именем этого компьютера. Поэтому в корне дерева стоит объект с именем компьютера, а ниже в иерархическом порядке располагаются все его зависимые объекты (логические и физические компоненты этого компьютера).

Таким образом, даже человеку, который не владеет глубокими знаниями о схеме классов WMI, в WMI Object Browser наглядно видно представление компьютера в качестве дерева его составных частей, причем каждую часть здесь можно детально изучить и произвести с ней манипуляции, не прибегая к помощи никаких дополнительных программных средств.

Напомним, что для запуска *WMI Object Browser, как и других административных утилит WMI*, нужно выбрать одноименный пункт в меню **Пуск | Программы | WMI Tools**, после чего подключиться к нужному пространству имен на локальном или удаленном компьютере.

Приложение 2

Полезные COM-объекты и примеры их использования

В *главе 10* мы работали в PowerShell с COM-объектами: создавали ярлык на рабочем столе с помощью объекта `WScript.Shell`, пользовались свойствами и методами серверов автоматизации Microsoft Word и Microsoft Excel.

При установке операционной системы Windows в ней регистрируется множество COM-объектов. Многие из них могут быть полезны администраторам и пользователям системы для автоматизации своих операций. Рассмотрим несколько примеров использования подобных объектов.

Управление проводником Windows с помощью объекта *Shell.Application*

Проводник Windows — это графическая оболочка, позволяющая управлять операционной системой с помощью различных диалоговых окон. Из оболочки (или сценария) PowerShell можно автоматизировать некоторые действия, выполняемые в проводнике Windows с помощью мыши или определенных комбинаций клавиш. Для этого нужно использовать COM-объект `Shell.Application`. Создадим экземпляр этого COM-объекта, сохранив ссылку на него в переменной `$shell`:

```
PS C:\> $shell=New-Object -com Shell.Application
```

С помощью командлета `Get-Member` посмотрим, какие методы и свойства имеет объект `$shell`:

```
PS C:\> $shell | Get-Member

   TypeName: System.__ComObject#{efd84b2d-4bcf-4298-be25-eb542a59fbda}

Name                 MemberType Definition
----                 ---------- ----------
AddToRecent          Method     void AddToRecent (Variant, string)
BrowseForFolder      Method     Folder BrowseForFolder (int, stri...
```

```
CanStartStopService  Method    Variant CanStartStopService (string)
CascadeWindows       Method    void CascadeWindows ()
ControlPanelItem     Method    void ControlPanelItem (string)
EjectPC              Method    void EjectPC ()
Explore              Method    void Explore (Variant)
ExplorerPolicy       Method    Variant ExplorerPolicy (string)
FileRun              Method    void FileRun ()
FindComputer         Method    void FindComputer ()
FindFiles            Method    void FindFiles ()
FindPrinter          Method    void FindPrinter (string, string,...
GetSetting           Method    bool GetSetting (int)
GetSystemInformation Method    Variant GetSystemInformation (str...
Help                 Method    void Help ()
IsRestricted         Method    int IsRestricted (string, string)
IsServiceRunning     Method    Variant IsServiceRunning (string)
MinimizeAll          Method    void MinimizeAll ()
NameSpace            Method    Folder NameSpace (Variant)
Open                 Method    void Open (Variant)
RefreshMenu          Method    void RefreshMenu ()
ServiceStart         Method    Variant ServiceStart (string, Var...
ServiceStop          Method    Variant ServiceStop (string, Vari...
SetTime              Method    void SetTime ()
ShellExecute         Method    void ShellExecute (string, Varian...
ShowBrowserBar       Method    Variant ShowBrowserBar (string, V...
ShutdownWindows      Method    void ShutdownWindows ()
Suspend              Method    void Suspend ()
TileHorizontally     Method    void TileHorizontally ()
TileVertically       Method    void TileVertically ()
ToggleDesktop        Method    void ToggleDesktop ()
TrayProperties       Method    void TrayProperties ()
UndoMinimizeALL      Method    void UndoMinimizeALL ()
Windows              Method    IDispatch Windows ()
WindowsSecurity      Method    void WindowsSecurity ()
Application          Property  IDispatch Application () {get}
Parent               Property  IDispatch Parent () {get}
```

Как можно понять из полученного списка, объект `Shell.Application` имеет довольно много методов для показа определенных окон Проводника Windows и управления открытыми окнами. Рассмотрим некоторые действия, которые позволяет выполнить данный объект. Мы предполагаем, что в переменной `$shell` уже сохранена ссылка на COM-объект `Shell.Application`.

Отображение специальных окон Проводника

Большое количество методов объекта `Shell.Application` позволяют открывать разнообразные специальные окна Проводника. Посмотрим, как вызывать наиболее часто встречающиеся из них.

Отображение определенной папки в Проводнике Windows

Метод `Explore` позволяет запустить новый экземпляр Проводника Windows, причем в нем сразу будет открыта папка, путь к которой указан в качестве параметра метода. Например, после выполнения следующей команды откроется новое окно Проводника Windows, в котором будет отображен корневой каталог диска c: (рис. П2.1):

```
PS C:\> $shell.Explore("c:\")
```

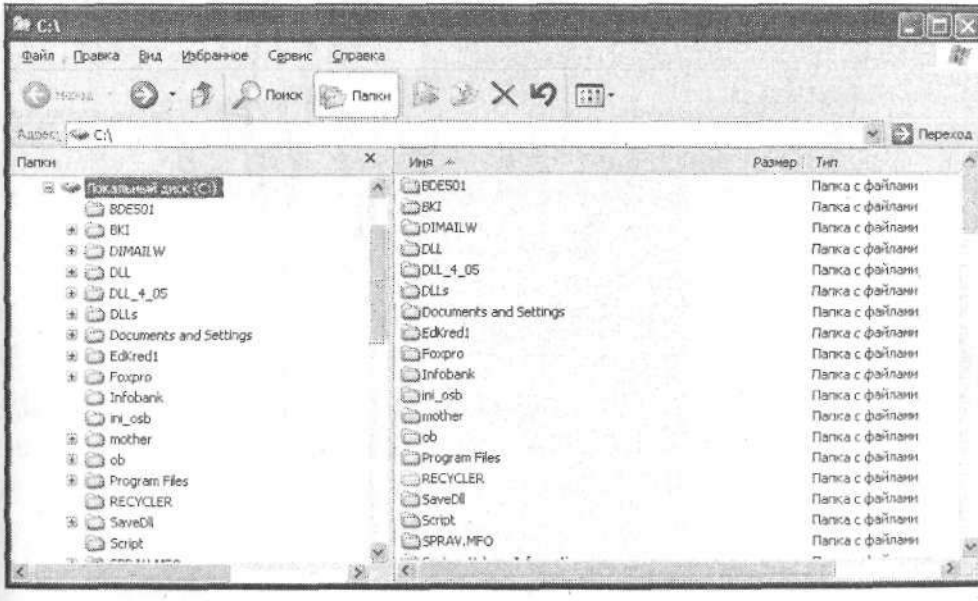


Рис. П2.1. Окно Проводника Windows для диска c:\

Вызов справочной системы Windows

Для вызова из оболочки PowerShell справочной системы Windows (рис. П2.2) нужно запустить метод `Help`:

```
PS C:\> $shell.Help()
```

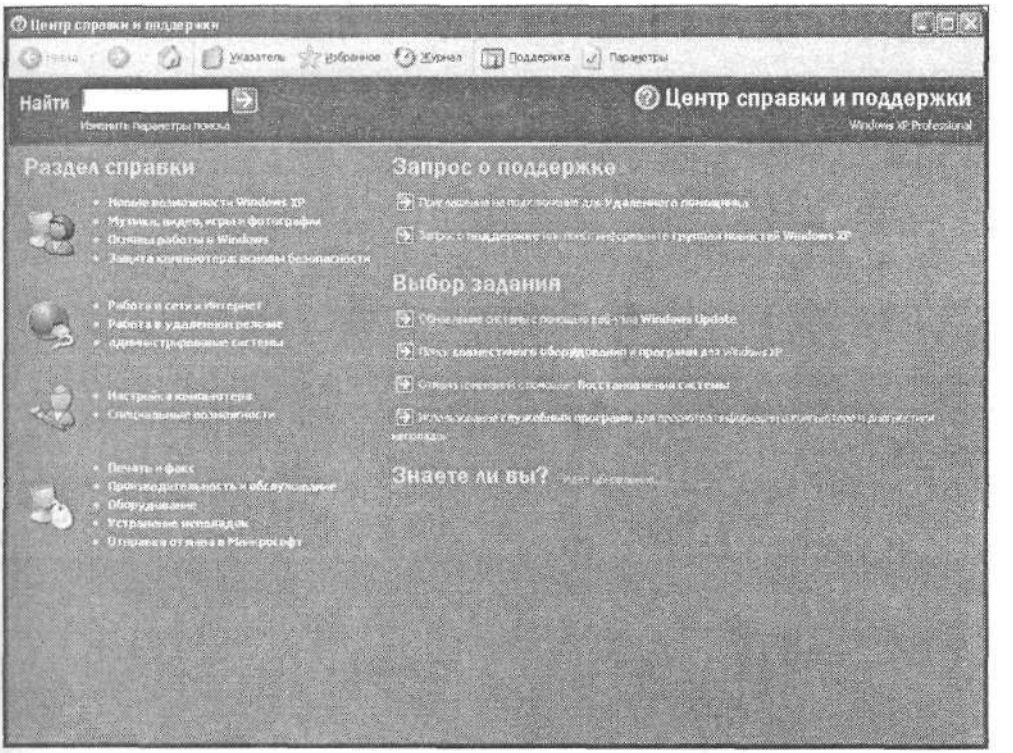


Рис. П2.2. Главное окно справочной системы Windows

Диалоговые окна поиска

Объект `Shell.Application` имеет три метода, с помощью которых можно открыть диалоговые окна для поиска файлов (метод `FindFiles`), компьютеров (метод `FindComputer`) или принтеров (метод `FindPrinter`).

Вот так открывается окно для поиска файлов (рис. П2.3):

```
PS C:\> $shell.FindFiles()
```

А так — окно для поиска компьютеров (рис. П2.4):

```
PS C:\> $shell.FindComputer()
```

И, наконец, так — окно для поиска принтеров (рис. П2.5):

```
PS C:\> $shell.FindPrinter()
```

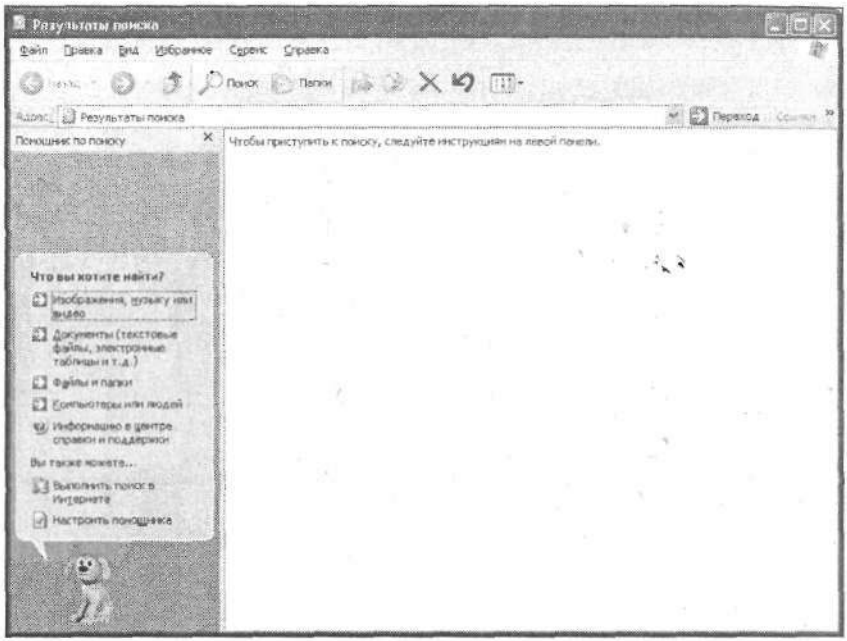


Рис. П2.3. Окно для поиска файлов

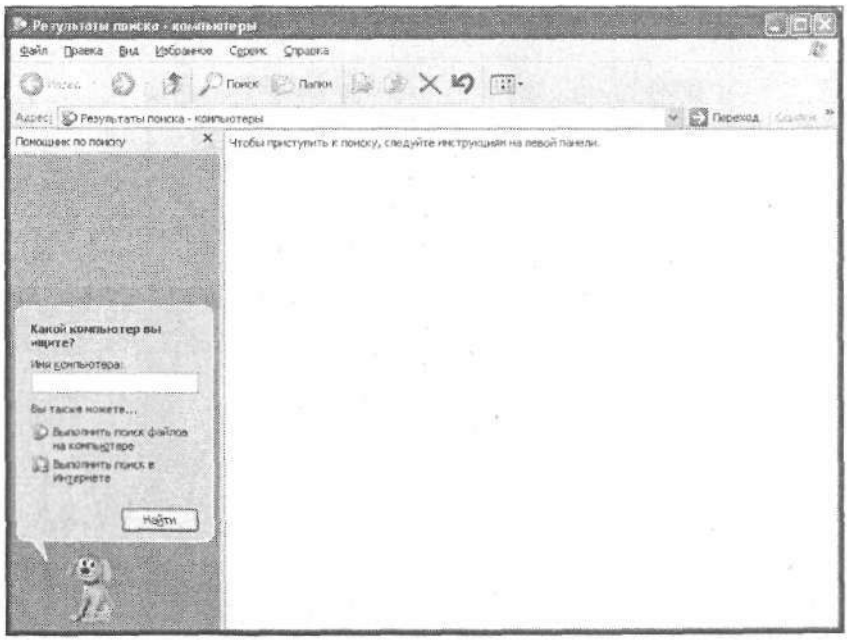


Рис. П2.4. Окно для поиска компьютеров

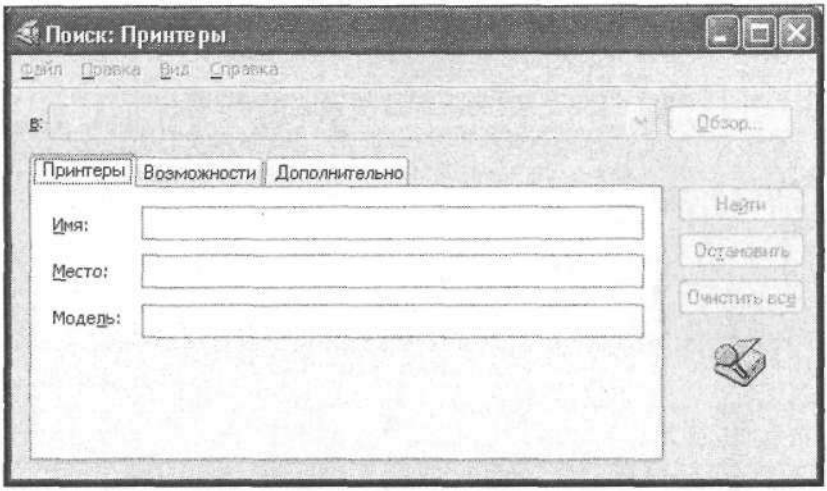


Рис. П2.5. Окно для поиска принтеров

Окно для запуска программ

С помощью метода `FileRun` можно из оболочки PowerShell открыть диалоговое окно для запуска программ (рис. П2.6):

```
PS C:\> $shell.FileRun()
```

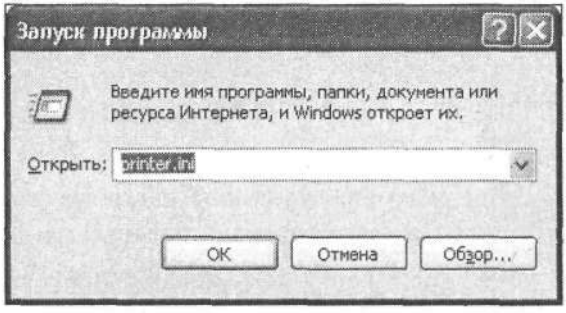


Рис. П2.6. Окно для запуска программ

Окно для установки времени и даты

Для отображения диалогового окна, позволяющего установить системную дату и время, нужно выполнить метод `SetTime` (рис. П2.7):

```
PS C:\> $shell.SetTime()
```

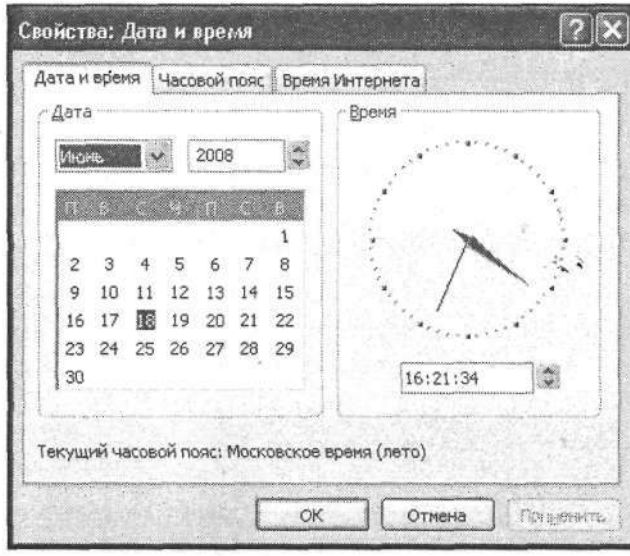


Рис. П2.7. Окно для установки даты и времени

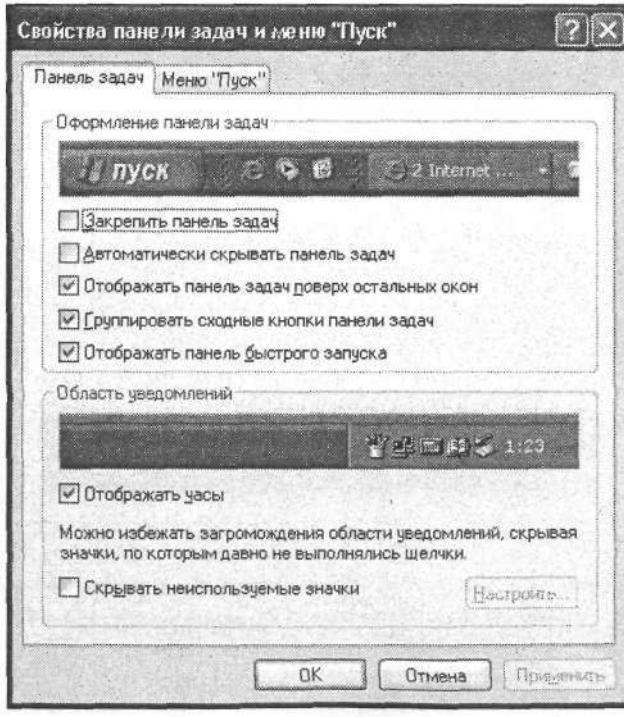


Рис. П2.8. Окно настройки панели задач и меню "Пуск"

Окно настройки панели задач

С помощью метода `TrayProperties` можно из оболочки PowerShell открыть диалоговое окно настройки панели задач (рис. П2.8):

```
PS C:\> $shell.TrayProperties()
```

Вызов элементов панели управления

Метод `ControlPanelItem` объекта `Shell.Application` позволяет открыть определенный элемент панели управления. Например, после выполнения следующих команд откроется диалоговое окно для настройки параметров экрана (рис. П2.9):

```
PS C:\> $shell.ControlPanelItem("desk.cpl")
```

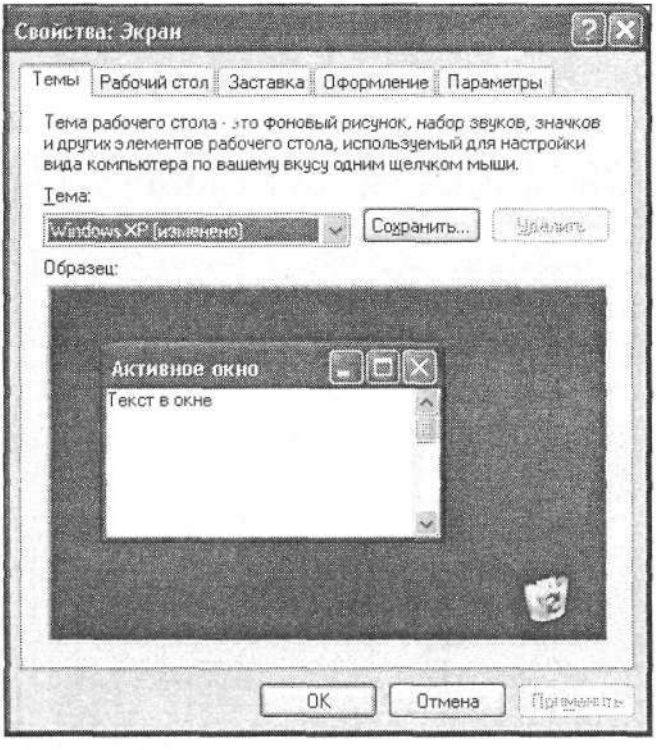


Рис. П2.9. Окно настройки параметров экрана

В качестве параметра метода `ControlPanelItem` указывается имя cpl-файла, соответствующего нужному элементу панели управления. Просмотреть спи-

сок всех cpl-файлов, имеющихся на локальном компьютере, можно с помощью следующей команды:

```
PS C:\> dir $env:windir\system32 -Recurse -Include *.cpl

   Каталог: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\system32

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          17.08.2004     16:05      69632 access.cpl
-a---          22.07.2005     14:56   18763776 ALSNDMGR.CPL
-a---          17.08.2004     16:05     551424 appwiz.cpl
-a---          10.10.1998      5:01     183808 BDEADMIN.CPL
-----          17.08.2004     16:05     110592 bthprops.cpl
-a---          17.08.2004     16:05     136704 desk.cpl
-----          17.08.2004     16:05      80384 firewall.cpl
-a---          17.08.2004     16:05     156160 hdwwiz.cpl
-a---          19.07.2005     11:08      77824 igfxcpl.cpl
-a---          17.08.2004     16:05     359424 inetcpl.cpl
-a---          17.08.2004     16:05     132096 intl.cpl
-----          17.08.2004     16:05     380416 irprops.cpl
-a---          17.08.2004     16:05      69120 joy.cpl
-a---          20.10.2001     19:00     188928 main.cpl
-a---          17.08.2004     16:05     620032 mmsys.cpl
-ar--          24.01.2001      5:05     121856 Mp3cnfg.cpl
-a---          20.10.2001     19:00      35840 ncpa.cpl
-----          17.08.2004     16:05      25600 netsetup.cpl
-a---          17.08.2004     16:05     259584 nusrmgr.cpl
-a---          20.10.2001     19:00      36864 nwc.cpl
-a---          17.08.2004     16:05      36864 odbccp32.cpl
-a---          17.08.2004     16:05     115712 powercfg.cpl
-a---          17.08.2004     16:05     300032 sysdm.cpl
-a---          20.10.2001     19:00      28160 telephon.cpl
-a---          17.08.2004     16:05      93696 timedate.cpl
-----          17.08.2004     16:05     148480 wscui.cpl
-----          17.08.2004     16:05     162304 wuaucpl.cpl
```

Замечание

Элементы панели управления можно вызывать и более простым способом, написав в командной строке PowerShell полное имя cpl-файла.

Управление открытыми окнами

Несколько методов объекта `Shell.Application` позволяют управлять расположением на рабочем столе открытых окон приложений.

Например, можно свернуть все открытые окна с помощью метода `MinimizeAll`:

```
PS C:\> $shell.MinimizeAll()
```

Восстановить свернутые окна на рабочем столе позволяет метод `UndoMinimizeAll`:

```
PS C:\> $shell.UndoMinimizeAll()
```

Методы `TileHorizontally` и `TileVertically` позволяют упорядочить открытые окна по горизонтали и вертикали, соответственно:

```
PS C:\> $shell.TileHorizontally()
PS C:\> $shell.TileVertically()
```

Метод `Windows` позволяет получить доступ к коллекции окон, открытых в Проводнике Windows и браузере Internet Explorer (то есть к окнам, открытым приложениями explorer.exe и iexplore.exe). Следующая команда определяет количество таких окон:

```
PS C:\> ($shell.Windows()).Count
5
```

Посмотрим, какие свойства и методы доступны для элементов данной коллекции:

```
PS C:\> $shell.Windows() | Get-Member

   TypeName: System.__ComObject#{d30c1661-cdaf-11d0-8a3e-00c04fc9e26e}

Name                MemberType Definition
----                ---------- ----------
ClientToWindow      Method     void ClientToWindow (int, int)
ExecWB              Method     void ExecWB (OLECMDID, OLECMDEXEC...
GetProperty         Method     Variant GetProperty (string)
GoBack              Method     void GoBack ()
GoForward           Method     void GoForward ()
GoHome              Method     void GoHome ()
GoSearch            Method     void GoSearch ()
Navigate            Method     void Navigate (string, Variant, V...
Navigate2           Method     void Navigate2 (Variant, Variant,...
PutProperty         Method     void PutProperty (string, Variant)
QueryStatusWB       Method     OLECMDF QueryStatusWB (OLECMDID)
```

```
Quit                    Method     void Quit ()
Refresh                 Method     void Refresh ()
Refresh2                Method     void Refresh2 (Variant)
ShowBrowserBar          Method     void ShowBrowserBar (Variant, Var...
Stop                    Method     void Stop ()
AddressBar              Property   bool AddressBar () {get} {set}
Application             Property   IDispatch Application () {get}
Busy                    Property   bool Busy () {get}
Container               Property   IDispatch Container () {get}
Document                Property   IDispatch Document () {get}
FullName                Property   string FullName () {get}
FullScreen              Property   bool FullScreen () {get} {set}
Height                  Property   int Height () {get} {set}
HWND                    Property   int HWND () {get}
Left                    Property   int Left () {get} {set}
LocationName            Property   string LocationName () {get}
LocationURL             Property   string LocationURL () {get}
MenuBar                 Property   bool MenuBar () {get} {set}
Name                    Property   string Name () {get}
Offline                 Property   bool Offline () {get} {set}
Parent                  Property   IDispatch Parent () {get}
Path                    Property   string Path () {get}
ReadyState              Property   tagREADYSTATE ReadyState () {get}
RegisterAsBrowser       Property   bool RegisterAsBrowser () {get} {...
RegisterAsDropTarget    Property   bool RegisterAsDropTarget () {get...
Resizable               Property   bool Resizable () {get} {set}
Silent                  Property   bool Silent () {get} {set}
StatusBar               Property   bool StatusBar () {get} {set}
StatusText              Property   string StatusText () {get} {set}
TheaterMode             Property   bool TheaterMode () {get} {set}
ToolBar                 Property   int ToolBar () {get} {set}
Top                     Property   int Top () {get} {set}
TopLevelContainer       Property   bool TopLevelContainer () {get}
Type                    Property   string Type () {get}
Visible                 Property   bool Visible () {get} {set}
Width                   Property   int Width () {get} {set}
```

Как видим, в нашем распоряжении имеется несколько методов для управления окнами Проводника Windows и браузера Internet Explorer (о действиях, выполняемых этими методами, можно догадаться по их названиям). Кроме того, можно изменять значения свойств, для которых указан признак `{set}`.

Для выделения из коллекции отдельного окна нужно воспользоваться методом `Item`, в качестве аргумента которого указывается индекс нужного окна (*нумерация начинается с нуля*). Например:

```
PS C:\> $win=$shell.Windows().Item(0)
PS C:\> $win

Application          : System.__ComObject
Parent               : System.__ComObject
Container            :
Document             : System.__ComObject
TopLevelContainer    : True
Type                 :
Left                 : 156
Top                  : 258
Width                : 800
Height               : 600
LocationName         : PS
LocationURL          : file:///C:/PS
Busy                 : False
Name                 : Microsoft Internet Explorer
HWND                 : 66522
FullName             : C:\WINDOWS\Explorer.EXE
Path                 : C:\WINDOWS\
Visible              : True
StatusBar            : False
StatusText           :  :
ToolBar              : 1
MenuBar              : True
FullScreen           : False
ReadyState           : 4
Offline              : False
Silent               : False
RegisterAsBrowser    : False
RegisterAsDropTarget : True
TheaterMode          : False
AddressBar           : True
Resizable            : False
```

В нашем случае первое окно было открыто с помощью Проводника Windows (в поле `FullName` хранится путь к файлу Explorer.exe), в нем отображается

содержимое папки C:\PS (значение свойства LocationURL). Модифицируя значения свойств объекта $win, можно открывать в данном окне другую папку или сетевой ресурс, а также менять размер или расположение окна. Например, для разворачивания окна на весь экран можно выполнить следующую команду:

```
PS C:\> $win.FullScreen=$True
```

Использование объектов Windows Script Host

Напомним еще раз, что сервер сценариев Windows Script Host (WSH) до появления оболочки PowerShell являлся основным инструментом автоматизации в Windows. Собственная объектная модель WSH представлена несколькими COM-объектами, которыми можно пользоваться из оболочки и сценариев PowerShell. Мы рассмотрим примеры использования некоторых методов и свойств объектов WScript.Network и WScript.Shell.

Работа с ресурсами локальной сети (объект *WScript.Network*)

Стандартным объектом WSH, позволяющим выполнять типовые операции с локальной сетью, является WScript.Network. С помощью этого объекта можно:

- узнать сетевое имя компьютера, имя текущего пользователя и название домена, в котором он зарегистрировался;
- получить список всех сетевых дисков и всех сетевых принтеров, подключенных к рабочей станции;
- подключить или отключить сетевой диск и принтер;
- установить сетевой принтер в качестве принтера, используемого по умолчанию.

Создадим экземпляр класса WScript.Network, сохранив ссылку на него в переменной $nw:

```
PS C:\> $nw=New-Object -com WScript.Network
```

Проверим с помощью командлета Get-Member, какие свойства и методы имеет полученный объект:

```
PS C:\> $nw | Get-Member

   TypeName: System.__ComObject#{24be5a31-edfe-11d2-b933-00104b365c9f}
```

```
Name                          MemberType Definition
----                          ---------- ----------
AddPrinterConnection          Method     void AddPrinterConnection ...
AddWindowsPrinterConnection   Method     void AddWindowsPrinterConn...
EnumNetworkDrives             Method     IWshCollection EnumNetwork...
EnumPrinterConnections        Method     IWshCollection EnumPrinter...
MapNetworkDrive               Method     void MapNetworkDrive (stri...
RemoveNetworkDrive            Method     void RemoveNetworkDrive (s...
RemovePrinterConnection       Method     void RemovePrinterConnecti...
SetDefaultPrinter             Method     void SetDefaultPrinter (st...
ComputerName                  Property   string ComputerName () {get}
Organization                  Property   string Organization () {get}
Site                          Property   string Site () {get}
UserDomain                    Property   string UserDomain () {get}
UserName                      Property   string UserName () {get}
UserProfile                   Property   string UserProfile () {get}
```

Как видим, в свойствах объекта `WScript.Network` хранится, в частности, информация об именах работающего пользователя (свойство `UserName`) и локального компьютера (свойство `ComputerName`), а также о домене, в котором зарегистрировался пользователь:

```
PS C:\> $nw

UserDomain   : SBRM
UserName     : Popov-AV
UserProfile  :
ComputerName : 404-POPOV
Organization :
Site         :
```

Методы объекта `WScript.Network` кратко описаны в табл. П2.1.

Таблица П2.1. *Методы объекта* `WScript.Network`

Метод	Описание
`AddPrinterConnection(`*`strLocalName, strRemoteName [,bUpdateProfile] [,strUser] [,strPassword]`*`)`	Подключает локальный порт компьютера к сетевому принтеру
`AddWindowsPrinterConnection(`*`strPrnPath`*`)`	Регистрирует принтер в Windows и подключает его к сетевому ресурсу. В отличие от `AddPrinterConnection`, этот метод позволяет создать связь с сетевым принтером без явного перенаправления вывода в локальный порт

Таблица П2.1 (окончание)

Метод	Описание
`EnumNetworkDrives()`	Возвращает коллекцию, в которой хранятся буквы и сетевые пути ко всем подключенным сетевым дискам
`EnumPrinterConnections()`	Возвращает коллекцию, в которой хранятся данные обо всех подключенных сетевых принтерах
`MapNetworkDrive(strLocalName, strRemoteName, [bUpdateProfile], [strUser], [strPassword])`	Подключает сетевой ресурс *strRemoteName* под локальным именем диска *strLocalName*
`RemoveNetworkDrive(strName, [bForce], [bUpdateProfile])`	Отключает подключенный сетевой диск
`RemovePrinterConnection(strName, [bForce], [bUpdateProfile])`	Отключает подключенный сетевой принтер
`SetDefaultPrinter(strPrinterName)`	Делает заданный сетевой принтер принтером по умолчанию

Приведем примеры использования методов объекта `WScript.Network`. Мы полагаем, что ссылка на экземпляр этого объекта содержится в переменной `$nw`.

Получение списка подключенных сетевых дисков и принтеров

С помощью методов `EnumNetworkDrives` и `EnumPrinterConnections` можно создать коллекции, содержащие, соответственно, сведения обо всех подключенных к локальной станции сетевых дисках и сетевых принтерах. Эти коллекции устроены следующим образом: первым элементом является буква диска или название порта, вторым — сетевое имя ресурса, с которым связан этот диск или принтер. Та же последовательность сохраняется для всех элементов коллекции.

Вызовем метод `EnumNetworkDrives`:

```
PS C:\> $nw.EnumNetworkDrives()
J:
\\file_server\sok$
K:
\\file_server\p_disk$
L:
\\file_server\bnk$
```

Как видим, в полученной коллекции чередуются буквы диска и их сетевые имена. Размер коллекции (то есть удвоенное число сетевых дисков) возвращает метод `Count`, а чтобы обращаться к ее элементам, нужно использовать метод `Item`, передавая ему требуемый индекс (нумерация начинается с нуля):

```
PS C:\> $nw_drives=$nw.EnumNetworkDrives()
PS C:\> $nw_drives.Count()
6
PS C:\> $nw_drives.Item(0)
J:
PS C:\> $nw_drives.Item(1)
\\file_server\sok$
PS C:\> $nw_drives.Item(2)
K:
PS C:\> $nw_drives.Item(3)
\\file_server\p_disk$
```

Вызовем теперь метод `EnumPrinterConnections`:

```
PS C:\> $nw.EnumPrinterConnections()
LPT1
\\Server1\Epson
IP_10.168.1.1
Xerox Phaser 3500 PCL 6
BIPORT
PaperPort Color
BIPORT
PaperPort
FILE:
Generic / Text Only
```

Здесь в возвращаемой коллекции чередуются названия локальных портов и сетевые имена принтеров, связанные с этими портами.

Подключение и отключение сетевых дисков

Имеющиеся в локальной сети общедоступные ресурсы (диски и принтеры) можно посредством методов объекта `WScript.Network` подключить к рабочей станции для совместного использования. При этом подключаемому сетевому диску нужно поставить в соответствие незанятую букву локального диска (например, если в системе уже имеются локальные или сетевые диски C:, D: и E:, то новый сетевой диск можно подключить под буквой F: или K:, но не E:).

Сетевой диск подключается с помощью метода `MapNetworkDrive` (см. табл. П2.1). Если необязательный параметр *`bUpdateProfile`* равен `True`, то создаваемое сетевое подключение будет сохранено в профиле пользователя.

Параметры *`strUser`* (имя пользователя) и *`strPassword`* (пароль) нужны в том случае, когда вы подключаете сетевой диск от имени пользователя, отличного от текущего пользователя, зарегистрированного в системе.

В следующем примере диск Z: подключается к сетевому ресурсу \\Server1\Programs:

```
PS C:\> $nw.MapNetworkDrive("Z:","\\Server1\Programs")
```

Отключить подключенный сетевой диск можно с помощью метода `RemoveNetworkDrive` (см. табл. П2.1). В качестве параметра *`strName`* должно быть указано либо локальное имя (буква сетевого диска), либо сетевое имя (имя подключенного сетевого ресурса); это зависит от того, каким образом осуществлялось подключение. Если сетевому ресурсу сопоставлена буква локального диска, то параметр *`strName`* должен быть локальным именем. Если сетевому ресурсу не сопоставлена никакая буква, то параметр *`strName`* должен быть сетевым именем.

Если необязательный параметр *`bForce`* равен `True`, то отключение сетевого ресурса будет произведено вне зависимости от того, используется этот ресурс в настоящее время или нет.

Если необязательный параметр *`bUpdateProfile`* равен `True`, то отключаемое сетевое подключение будет удалено из профиля пользователя.

В следующем примере производится подключение диска Z: к сетевому ресурсу, а затем отключение этого ресурса:

```
PS C:\> $nw.MapNetworkDrive("Z:","\\Server1\Programs")
PS C:\> $nw.RemoveNetworkDrive("Z:")
```

Подключение и отключение сетевых принтеров

В случае подключения сетевого принтера можно либо напрямую соединиться с этим принтером (для печати из приложений Windows), либо поставить в соответствие удаленному принтеру локальный порт (для печати из старых приложений MS-DOS).

Для подключения принтера к локальному порту компьютера используется метод `AddPrinterConnection` (см. табл. П2.1). Если необязательный параметр *`bUpdateProfile`* равен `True`, то создаваемое сетевое подключение будет сохранено в профиле пользователя.

Параметры *strUser* (имя пользователя) и *strPassword* (пароль) нужны в том случае, когда вы подключаете сетевой принтер от имени пользователя, отличного от зарегистрированного в данный момент в системе.

В следующем примере метод AddPrinterConnection применяется для подключения принтера с сетевым именем \\Server1\Epson к локальному порту LPT1:

```
PS C:\> $nw.AddPrinterConnection("LPT1", "\\Server1\Epson")
```

Для создания связи с сетевым принтером без явного перенаправления вывода в локальный порт применяется метод AddWindowsPrinterConnection, например:

```
PS C:\> $nw.AddWindowsPrinterConnection("\\printserv\DefaultPrinter")
```

Для отключения подключенного сетевого принтера используется метод RemovePrinterConnection (см. табл. П2.1). В качестве параметра *strName* должно быть указано либо локальное имя (название порта), либо сетевое имя (имя подключенного сетевого принтера); это зависит от того, каким образом осуществлялось подключение. Если сетевому ресурсу явным образом сопоставлен локальный порт (например, LPT1), то параметр *strName* должен быть локальным именем. Если сетевому принтеру локальный порт не сопоставлен, то параметр *strName* должен быть сетевым именем.

Параметры *bForce* и *bUpdateProfile* в этом методе имеют то же значение, что и одноименные параметры в методе RemoveNetworkDrive.

В следующем примере отключается сетевой принтер, который был назначен на порт LPT1:

```
PS C:\> $nw.RemovePrinterConnection("LPT1:")
```

Установка принтера по умолчанию

Метод SetDefaultPrinter делает указанный сетевой принтер принтером по умолчанию.

В следующем примере с помощью метода AddPrinterConnection к порту LPT1: подключается сетевой принтер \\Server1\Epson, который затем устанавливается принтером по умолчанию:

```
PS C:\> $nw.AddPrinterConnection("LPT1", "\\Server1\Epson")
PS C:\> $nw.SetDefaultPrinter("\\Server1\Epson")
```

Вывод информационного окна (объект *WScript.Shell*)

Объект WScript.Shell имеет метод Popup(*strText, [nSecToWait], [strTitle], [nType]*), который выводит на экран информационное окно с сообщением.

заданным параметром `strText`. Параметр `nSecToWait` задает количество се кунд, по истечении которых окно будет автоматически закрыто, параметр `strTitle` определяет заголовок окна, параметр `nType` указывает тип кнопок и значка для окна. Если параметр `strTitle` не задан, то по умолчанию заголовком окна будет "Windows Script Host."

Параметр `nType` может принимать те же значения, что и в функции `MessageBox` из Microsoft Win32 API. В табл. П2.2 описаны некоторые возможные значения параметра `nType` и их смысл (полный список значений этого параметра можно посмотреть в описании функции `MessageBox` в документации по функциям Windows API).

Таблица П2.2. *Типы кнопок и иконок для метода* `Popup`

Значение `nType`	**Константа Visual Basic**	**Описание**
0	`vbOkOnly`	Выводится кнопка **OK**
1	`vbOkCancel`	Выводятся кнопки **OK** и **Отмена** (Cancel)
2	`vbAbortRetryIgnore`	Выводятся кнопки **Стоп** (Abort), **Повтор** (Retry) и **Пропустить** (Ignore)
3	`vbYesNoCancel`	Выводятся кнопки **Да** (Yes), **Нет** (No) и **Отмена** (Cancel)
4	`vbYesNo`	Выводятся кнопки **Да** (Yes) и **Нет** (No)
5	`vbRetryCancel`	Выводятся кнопки **Повтор** (Retry) и **Отмена** (Cancel)
16	`vbCritical`	Выводится значок **Stop Mark**
32	`vbQuestion`	Выводится значок **Question Mark**
48	`vbExclamation`	Выводится значок **Exclamation Mark**
64	`vbInformation`	Выводится значок **Information Mark**

В методе `Popup` можно комбинировать значения параметра, приведенные в табл. П2.2. Например, в результате выполнения следующих команд:

```
PS C:\> $shell=New-Object -com WScript.Shell
PS C:\> $shell.Popup("Копирование завершено успешно",5,"Ура",65)
```

на экран будет выведено информационное окно, показанное на рис. П2.10, которое автоматически закроется через 5 секунд.

Метод `Popup` возвращает целое значение, с помощью которого можно узнать, какая именно кнопка была нажата для выхода (табл. П2.3).

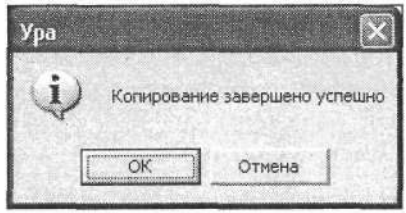


Рис. П2.10. Информационное окно, созданное методом `Popup`

Таблица П2.3. *Возвращаемые методом* `Popup` *значения*

Значение	Константа Visual Basic	Описание
–1		Пользователь не нажал ни на одну из кнопок в течение времени, заданного параметром `nSecToWait`
1	`vbOk`	Нажата кнопка **OK**
2	`vbCancel`	Нажата кнопка **Отмена** (Cancel)
3	`vbAbort`	Нажата кнопка **Стоп** (Abort)
4	`vbRetry`	Нажата кнопка **Повтор** (Retry)
5	`vbIgnore`	Нажата кнопка **Пропустить** (Ignore)
6	`vbYes`	Нажата кнопка **Да** (Yes)
7	`vbNo`	Нажата кнопка **Нет** (No)

Переключение между приложениями, имитация нажатий клавиш (объект *WScript.Shell*)

Производить переключение между окнами нескольких запущенных приложений позволяет метод `AppActivate` объекта `WScript.Shell`. В качестве аргумента этого метода нужно указывать либо заголовок активизируемого окна, либо программный идентификатор (PID) процесса, который запущен в этом окне. Предпочтительным является использование PID, который можно получить с помощью свойства `ProcessID` объектов, генерируемых при запуске процессов с помощью метода `Exec` объекта `WScript.Shell`. Использование заголовка окна в методе `AppActivate` имеет ряд недостатков:

- при написании сценария необходимо знать точное название заголовка;
- само приложение может изменить текст в заголовке окна;

- в случае наличия нескольких окон с одинаковыми заголовками AppActivat всегда будет активизировать один и тот же экземпляр, а доступ к други окнам получить не удастся.

Активизировав то или иное окно, в котором выполняется приложени Windows, можно из оболочки или сценария PowerShell сымитировать нажа тия клавиш в этом окне. Для этого используется метод `SendKeys(string` объекта `WshShell`.

> ***ЗАМЕЧАНИЕ***
>
> Для нормальной работы метода `SendKeys` необходимо, чтобы языком п умолчанию в операционной системе был назначен английский язык.

Каждая клавиша задается одним или несколькими символами. Например, дл того чтобы задать нажатие друг за другом букв А, Б и В, нужно указать качестве параметра для `SendKeys` строку "АВВ": *string*="АВВ".

Несколько символов имеют в методе `SendKeys` специальное значение: +, ^, % ~, (,). Для того чтобы задать один из этих символов, их нужно заключит в фигурные скобки ({}). Например, для задания знака плюс используется {+} Квадратные скобки ([]), хотя они и не имеют в методе `SendKey` специального смысла, также должны заключаться в фигурные скобки. Кром этого, для задания самих фигурных скобок следует использовать следующи конструкции: {{} (левая скобка) и {}} (правая скобка).

Для задания неотображаемых символов, таких как <Enter> или <Tab> и специальных клавиш в методе `SendKeys` используются коды, представленны в табл. П2.4.

Таблица П2.4. *Коды специальных клавиш для метода* `SendKey`

Клавиша	Код	Клавиша	Код
<Backspace>	{BACKSPACE}, {BS} или {BKSP}	<→>	{RIGHT}
<Break>	{BREAK}	<F1>	{F1}
<Caps Lock>	{CAPSLOCK}	<F2>	{F2}
<Del> или <Delete>	{DELETE} или {DEL}	<F3>	{F3}
<End>	{END}	<F4>	{F4}
<Enter>	{ENTER} или ~	<F5>	{F5}
<Esc>	{ESC}	<F6>	{F6}
<Home>	{HELP}	<F7>	{F7}

Таблица П2.4 *(окончание)*

Клавиша	Код	Клавиша	Код
<Ins> или <Insert>	{INSERT} или {INS}	<F8>	{F8}
<Num Lock>	{NUMLOCK}	<F9>	{F9}
<Page Down>	{PGDN}	<F10>	{F10}
<Page Up>	{PGUP}	<F11>	{F11}
<Print Screen>	{PRTSC}	<F12>	{F12}
<Scroll Lock>	{SCROLLLOCK}	<F13>	{F13}
<Tab>	{TAB}	<F14>	{F14}
<↑>	{UP}	<F15>	{F15}
<←>	{LEFT}	<F16>	{F16}
<↓>	{DOWN}		

Для задания комбинаций клавиш с <Shift>, <Ctrl> или <Alt>, перед соответствующей клавишей нужно поставить один или несколько кодов из табл. П2.5.

Таблица П2.5. *Коды клавиш <Shift>, <Ctrl> и <Alt>*

Клавиша	Код
<Shift>	+
<Ctrl>	^
<Alt>	%

Для того чтобы задать комбинацию клавиш, которую нужно набирать, удерживая нажатыми клавиши <Shift>, <Ctrl> или <Alt>, нужно заключить коды этих клавиш в скобки. Например, если требуется сымитировать нажатие клавиш G и S при нажатой клавише <Shift>, следует использовать последовательность "+(GS)". Для того же, чтобы задать одновременное нажатие клавиш <Shift>+<G>, а затем <S> (уже без <Shift>), используется "+GS".

В методе `SendKeys` можно упрощенным способом задать несколько нажатий подряд одной и той же клавиши. Для этого необходимо в фигурных скобках указать код нужной клавиши, а через пробел — число нажатий. Например, {LEFT 42} означает нажатие клавиши <←> 42 раза подряд; {h 10} означает нажатие клавиши h 10 раз подряд.

Замечание

Метод SendKeys не может быть использован для имитации нажатий клавиш в приложениях, которые не были разработаны специально для запуска в Microsoft Windows (например, для приложений MS-DOS).

Рассмотрим пример использования метода SendKeys. Создадим сначала экземпляр класса WScript.Shell и сохраним его в переменной $shell:

```
PS C:\> $shell=New-Object -com WScript.Shell
```

С помощью метода Exec запустим Калькулятор (calc.exe), сохранив возвращаемый объект в переменной $calc:

```
PS C:\> $calc=$shell.Exec("calc")
```

После того как окно Калькулятора появится на экране, активизируем это приложение с помощью метода AppActivate и с помощью SendKeys последовательно пошлем в его окно нажатия клавиш <1>, <+>, <2> и <Enter> (для наглядности после каждого вызова метода SendKeys мы делаем паузу в 1 секунду, используя командлет Start-Sleep). После этого результат вычислений (символ "3") скопируем в буфер Windows, сымитировав нажатие клавиш <Ctrl>+<C>:

```
PS C:\> if ($shell.AppActivate($calc.ProcessID)) {
>>    $shell.SendKeys("1{+}")
>>    Start-Sleep 1
>>    $shell.SendKeys("2")
>>    Start-Sleep 1
>>    $shell.SendKeys("~")
>>    Start-Sleep 1
>>    $shell.SendKeys("^c");
>> }
>>
```

Выведем на экран сообщение о том, что Калькулятор будет закрыт:

```
PS C:\> [void]$shell.Popup("Закрываем калькулятор")
```

После этой команды окно Калькулятора теряет фокус. Чтобы вновь активизировать это окно, используем метод AppActivate, передав ему PID Калькулятора, после чего для закрытия окна Калькулятора имитируем нажатие клавиш <Alt>+<F4>:

```
PS C:\> if ($shell.AppActivate($calc.ProcessID)) {
>>    $shell.SendKeys("%{F4}")
>> }
>>
```

Закрыв Калькулятор, запускаем Блокнот (notepad.exe) и записываем результаты работы Калькулятора (вставка вычисленной суммы из буфера производится с помощью "нажатия" <Ctrl>+<V>):

```
PS C:\> $n=$shell.Exec("notepad")
PS C:\> Start-Sleep 1
PS C:\> if ($shell.AppActivate($n.ProcessID)) {
>>   $shell.SendKeys("1{+}2=")
>>   $shell.SendKeys("^v")
>>   $shell.SendKeys(" {(}c{)} Calculator")
>> }
>>
```

В результате в Блокноте отобразится текст, показанный на рис. П2.11.

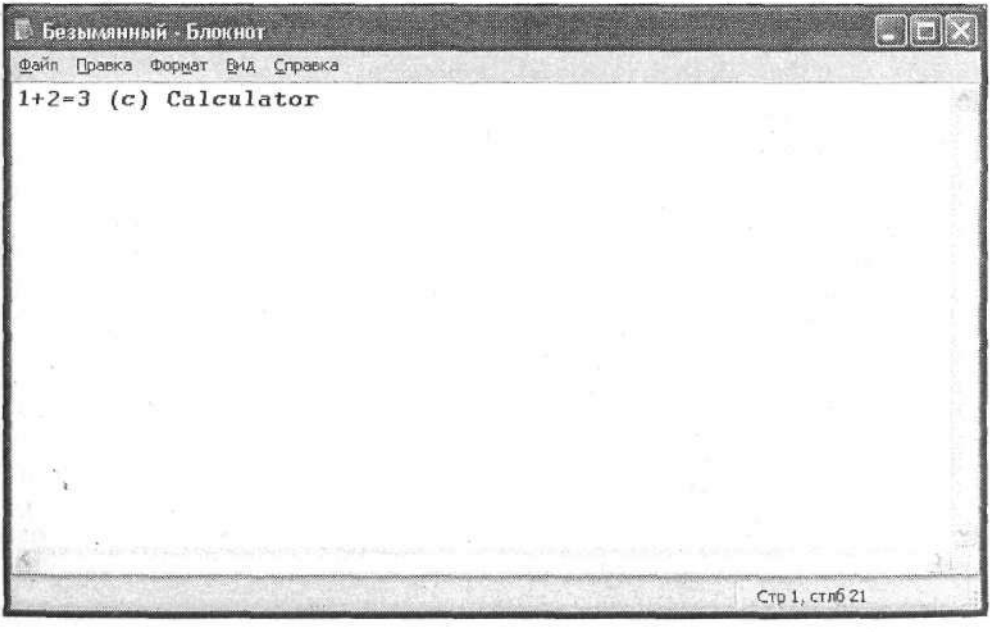


Рис. П2.11. Результат имитации нажатий клавиш в Блокноте

Полностью сценарий SendKeys.ps1, имитирующий нажатия клавиш, приведен в листинге П2.1.

Листинг П2.1. Имитация нажатий клавиш и обмен данными между приложениями (файл SendKeys.ps1)

```
################################################
# Имя: SendKeys.ps1
# Язык: PoSH 1.0
# Описание: Имитация нажатий клавиш и обмен
# данными между приложениями через буфер Windows
################################################
# Создаем экземпляр класса WScript.Shell
$shell=New-Object -com WScript.Shell

# Запускаем процесс calc.exe (Калькулятор)
$calc=$shell.Exec("calc")

# Приостанавливаем выполнение сценария, чтобы
# окно Калькулятора успело появиться на экране
Start-Sleep 1

# Активизируем окно Калькулятора
if ($shell.AppActivate($calc.ProcessID)) {
  # Посылаем нажатия клавиш в окно Калькулятора
  $shell.SendKeys("1{+}")
  Start-Sleep 1
  $shell.SendKeys("2")
  Start-Sleep 1
  $shell.SendKeys("~")
  Start-Sleep 1
  # Копируем результат вычисления в буфер Windows (<Ctrl>+C)
  $shell.SendKeys("^c");
}

# Выводим сообщение (активное окно меняется)
[void]$shell.Popup("Закрываем калькулятор")

# Вновь активизируем окно Калькулятора
if ($shell.AppActivate($calc.ProcessID)) {
  # Закрываем окно Калькулятора (<Alt>+<F4>)
  $shell.SendKeys("%{F4}")
```

```
}

# Запускаем процесс notepad.exe (Блокнот)
$n=$shell.Exec("notepad")

# Приостанавливаем выполнение сценария, чтобы
# окно Блокнота успело появиться на экране
Start-Sleep 1
# Активизируем окно Блокнота
if ($shell.AppActivate($n.ProcessID)) {
  # Посылаем нажатия клавиш в окно Блокнота
  $shell.SendKeys("1{+}2=")
  # Вставляем содержимое буфера Windows (<Ctrl>+V)
  $shell.SendKeys("^v")
  $shell.SendKeys(" {(}c{)} Calculator")
}
########## Конец сценария ##################################
```

Доступ к специальным папкам Windows (объект *WScript.Shell*)

При регистрации нового пользователя в системе для него автоматически создаются несколько специальных папок (например, папка для Рабочего стола (Desktop) и папка для меню **Пуск**), путь к которым впоследствии может быть тем или иным способом изменен. Пути к таким папкам может быть нужно знать, например, для автоматического создания ярлыков на рабочем столе.

С помощью свойства `SpecialFolders` объекта `WScript.Shell` можно получить доступ к коллекции, содержащей пути к таким специальным папкам. Для этого нужно воспользоваться их стандартизированными названиями, которые не зависят от локализации операционной системы и прочих ее настроек. В табл. П2.6 перечислены такие названия.

Таблица П2.6. *Специальные папки активного пользователя*

Стандартизированное название папки	Описание
Desktop	Рабочий стол
Favorites	Избранное
Fonts	Папка шрифтов

Таблица П2.6 *(окончание)*

Стандартизированное название папки	Описание
MyDocuments	Мои документы
NetHood	Сетевое окружение
PrintHood	Сетевые принтеры
Programs	Ярлыки к программам, появляющимся в пункте **Все программы** меню **Пуск**
Recent	Ярлыки к недавно использованным документам
SendTo	Ярлыки, отображающиеся в пункте **Отправить** контекстного меню файлов в Проводнике
StartMenu	Программы, показываемые в меню **Пуск**
Startup	Автоматически запускаемые при загрузке файлы
Templates	Шаблоны новых документов

Кроме того, имеются еще четыре папки, которые хранят данные для всех пользователей. Они перечислены в табл. П2.7.

Таблица П2.7. *Общие для всех пользователей специальные папки*

Стандартизированное название папки	Описание
AllUsersDesktop	Рабочий стол
AllUsersStartMenu	Меню **Пуск**
AllUsersPrograms	Ярлыки к программам, отображаемым в пункте **Все программы** меню **Пуск**
AllUsersStartup	Ярлыки к файлам автозапуска

Рассмотрим пример. Создадим экземпляр класса `WScript.Shell` (переменная `$shell`):

```
PS C:\> $shell=New-Object -com WScript.Shell
```

В переменной `$fldrs` сохраним ссылку на коллекцию специальных папок:

```
PS C:\> $fldrs=$shell.SpecialFolders
```

Проверим содержимое коллекции `$fldrs`:

```
PS C:\> $fldrs
C:\Documents and Settings\All Users\Рабочий стол
```

```
C:\Documents and Settings\All Users\Главное меню
C:\Documents and Settings\All Users\Главное меню\Программы
C:\Documents and Settings\All Users\Главное меню\Программы\Автозагрузка
C:\Documents and Settings\Popov\Рабочий стол
C:\Documents and Settings\Popov\Application Data
C:\Documents and Settings\Popov\PrintHood
C:\Documents and Settings\Popov\Шаблоны
C:\WINDOWS\Fonts
C:\Documents and Settings\Popov\NetHood
C:\Documents and Settings\Popov\Рабочий стол
C:\Documents and Settings\Popov\Главное меню
C:\Documents and Settings\Popov\SendTo
C:\Documents and Settings\Popov\Recent
C:\Documents and Settings\Popov\Главное меню\Программы\Автозагрузка
C:\Documents and Settings\Popov\Избранное
C:\Documents and Settings\Popov\Мои документы
C:\Documents and Settings\Popov\Главное меню\Программы
```

Доступ к отдельному элементу данной коллекции производится либо через числовой индекс, либо через стандартизированное имя соответствующей папки (см. табл. П2.6 и П2.7). Например:

```
PS C:\> $fldrs.Item("desktop")
C:\Documents and Settings\Popov\Рабочий стол
PS C:\> $fldrs.Item("favorites")
C:\Documents and Settings\Popov\Избранное
PS C:\> $fldrs.Item("programs")
C:\Documents and Settings\Popov\Главное меню\Программы
```

Удаление некорректных ярлыков (объект *WScript.Shell*)

Ярлыки (файлы с расширением lnk) являются важной частью графического интерфейса пользователя Windows (например, главное меню полностью состоит из ярлыков). В *главе 10* мы рассматривали, каким образом можно создать ярлык с помощью методов объекта `WScript.Shell`. Однако если объект, на который ссылается ярлык, удалить или переместить, то соответствующий ярлык станет бесполезным. Давайте напишем сценарий, который будет находить и удалять такие некорректные ярлыки на рабочем столе.

Создадим экземпляр класса `WScript.Shell` (переменная `$shell`):

```
PS C:\> $shell=New-Object -com WScript.Shell
```

В переменной `$fldrs` сохраним ссылку на коллекцию специальных папок:

```
PS C:\> $fldrs=$shell.SpecialFolders
```

В переменную $desktop поместим путь к рабочему столу активного пользователя:

```
PS C:\> $desktop=$fldrs.Item("desktop")
```

Теперь с помощью конвейера командлетов `dir` (`Get-ChildItem`) и `ForEach-Object` переберем все файлы на рабочем столе с расширением lnk. Для каждого такого файла будем создавать с помощью метода `CreateShortcut` соответствующий объект-ярлык и проверять с помощью командлета `Test-Path` наличие объекта, на который ссылается этот ярлык (путь к этому объекту хранится в свойстве `TargetPath` объекта-ярлыка). Если объект не существует (командлет `Test-Path` возвращает `$False`), то проверяемый файл с расширением lnk удаляется:

```
PS C:\> dir "$desktop\*.lnk" | ForEach-Object{
>>   $shortcut=$shell.CreateShortcut($_.FullName);
>>   if (-not (Test-Path $shortcut.TargetPath)) {del $_.FullName};
>> }
>>
```

Полностью сценарий ClearLnk.ps1, удаляющий некорректные ярлыки с рабочего стола, приведен в листинге П2.2.

Листинг П2.2. Удаление некорректных ярлыков с рабочего стола (файл ClearLnk.ps1)

```
##############################################
# Имя: ClearLnk.ps1
# Язык: PoSH 1.0
# Описание: Удаление некорректных ярлыков
# с рабочего стола активного пользователя
##############################################
# Создаем экземпляр класса WScript.Shell
$shell=New-Object -com WScript.Shell

# Создаем коллекцию специальных папок
$fldrs=$shell.SpecialFolders
```

```
# Определяем путь к рабочему столу
$desktop=$fldrs.Item("desktop")

# Перебираем файлы с расширением lnk
dir "$desktop\*.lnk" | ForEach-Object{
  # Создаем объект-ярлык для текущего файла
  $shortcut=$shell.CreateShortcut($_.FullName);
  # Удаляем файл, для которого не существует целевой объект
  if (-not (Test-Path $shortcut.TargetPath)) {del $_.FullName};
  }
############ Конец сценария ##################################
```

Ссылки на ресурсы Интернета

Сайты компании Microsoft

Адрес	Описание
http://www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx или **http://microsoft.com/powershell**	*Официальный сайт* Windows PowerShell (на английском языке)
http://www.microsoft.com/technet/technetmag	Журнал "TechNet Magazine" публикует статьи по различным технологиям Microsoft, в том числе и по Windows PowerShell (имеется перевод на русский язык)
http://www.microsoft.com/technet/scriptcenter	Сайт "TechNet Script Center" содержит большое количество примеров сценариев PowerShell, а также ссылки *на статьи и вебкасты, посвященные* PowerShell (на английском языке)
http://msdn.microsoft.com/developer	Электронная библиотека MSDN (Microsoft Developer Network) содержит подробную информацию об объектных моделях, которые может использовать PowerShell (частично на русском, но в основном на английском языке)

Другие сайты

Адрес	Описание
http://www.script-coding.info/	Статьи и примеры сценариев по различным языкам сценариев (на русском языке)
http://forum.sysfaq.ru/	Форум системных администраторов, содержащий раздел по языкам сценариев и утилитам командной строки (на русском языке)

(окончание)

Адрес	Описание
http://www.scriptinganswers.com/	Статьи, примеры сценариев, ссылки на сайты схожей тематики (на английском языке)
http://www.powershellcommunity.org/	Документация, статьи, примеры сценариев, ответы на часто задаваемые вопросы (на английском языке)

Группы новостей

Адрес	Описание
http://www.microsoft.com/communities/newsgroups/en-us/?dg=microsoft.public.windows.powershell	Веб-интерфейс для группы новостей PowerShell (на английском языке)

Блоги

Адрес	Описание
http://blogs.msdn.com/powershell	Официальный блог разработчиков PowerShell. Здесь можно найти ссылки на множество других блогов, посвященных PowerShell (на английском языке)
http://dmitrysotnikov.wordpress.com/	Англоязычный блог Дмитрия Сотникова, одного из разработчиков AD Cmdlets (командлетов для работы с Active Directory) и PowerGUI (графическая оболочка для PowerShell)
http://www.itcommunity.ru/blogs/dmitrysotnikov	Русскоязычный блог Дмитрия Сотникова
http://xaegr.wordpress.com/	Русскоязычный блог Василия Гусева (Microsoft Most Valuable Professional по специализации Admin Frameworks)
http://blogs.technet.com/abeshkov	Русскоязычный блог Андрея Бешкова

Список литературы

1. Payette B. Windows PowerShell in Action. — Manning Publications Co, 2007.
2. Holmes L. Windows PowerShell Cookbook. — O'Reilly, 2007.
3. Wilson E. Windows PowerShell Step-by-Step. — Microsoft Press, 2007.
4. Watt A. Professional Windows PowerShell (Programmer to Programmer). — Wrox Press, 2007.
5. Kopczynski T. Windows PowerShell Unleashed. — SAMS Publishing, 2007.
6. Jones D., Hicks J. Windows PowerShell: TFM. — Sapien, 2006.
7. Cookey-Gam J., Keane B., Rosen J., Runyon J., Stidley J. Professional Windows PowerShell for Exchange Server 2007 Service Pack 1 (Programmer to Programmer). — Wrox Press, 2008.
8. Koch F. Windows PowerShell. — (Электронная версия книги доступна для бесплатного скачивания по адресу **http://blogs.technet.com/chitpro-de/archive/2007/05/10/english-version-of-windows-powershell-course-book-available-for-download.aspx**).
9. Koch F. Administrative tasks using PowerShell. — (Электронная версия книги доступна для бесплатного скачивания по адресу **http://blogs.technet.com/chitpro-de/archive/2008/02/28/free-windows-powershell-workbook-server-administration.aspx**).

Введение в Windows PowerShell

Попов
Андрей Владимирович,
кандидат физико-математических наук, доцент кафедры технологий программирования Мордовского государственного университета, автор книг «Командные файлы и сценарии Windows Script Host», «Windows Script Host для Windows 2000/XP», «Администрирование Windows с помощью WMI и WMIC».

АВТОМАТИЗАЦИЯ РЕШЕНИЯ ПОВСЕДНЕВНЫХ ЗАДАЧ АДМИНИСТРАТОРА

Книга предназначена для пользователей, программистов и администраторов Windows, которые хотят узнать о возможностях оболочки командной строки Windows PowerShell и использовать их для автоматизации повседневных задач администрирования. Описаны базовые механизмы и основные конструкции языка PowerShell, приведены приемы интерактивной работы в оболочке и создания сценариев. Даны примеры обращения к объектным моделям .NET, WMI, ADSI и COM. Рассмотрены варианты решения практических задач администрирования Windows с помощью PowerShell.

КАТЕГОРИЯ: операционные системы

ISBN 978-5-9775-0283-2

ИКС:Сист.админ.Введ.в Window

Цена:454р.00к.

9785977502832 13.11.08 502832

СИСАДМИН СИСТЕМНЫЙ АДМИНИСТРАТОР

bhv®

БХВ-ПЕТЕРБУРГ
194354, Санкт-Петербург, ул. Есенина, 5Б
E-mail: mail@bhv.ru
Internet: www.bhv.ru
Тел./факс: (812) 591-6243